The Log Analysis in an Automatic Approach

THE LOG ANALYSIS IN AN AUTOMATIC APPROACH

BY

JIANHUI LEI, B.A.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2012) McMaster University
(Computing and Software) Hamilton, Ontario, Canada

TITLE:	The Log Analysis in an Automatic Approach
AUTHOR:	Jianhui Lei B.A.Sc., (Computer Engineering) University of Toronto Toronto, Canada
SUPERVISOR:	Dr. Tom Maibaum
NUMBER OF PAGES:	xiv, 195

To my family

Abstract

Large software systems tend to have complex architecture and numerous lines of source code. As software systems have grown in size and complexity, it has become increasingly difficult to deliver bug-free software to end-users. Most system failures occurring at run-time are directly caused by system defects; therefore diagnosis of software defects becomes an important but challenging task in software development and maintenance.

A system log is one available source of information from a software system. Software developers have used system logs to record program variable values, trace execution, report run-time statistics and print out full-sentence messages. This makes system logs a helpful resource for diagnosis of software defects. The conventional log analysis requires human intervention to examine run-time information in system logs and to apply their expertise to software systems, in order to determine the root cause of a software defect and work out a concrete solution. Complex software systems can generate thousands of system logs in a relatively short time frame. Analyzing such large amounts of information turns out to be extremely time-consuming. Automated techniques are needed to improve the efficiency and quality of the diagnostic process.

This thesis presents an automated approach to diagnosis of software defects, combining source code analysis, log analysis and sequential pattern mining, to detect anomalies among system logs, diagnose reported system errors and narrow down the range of source code lines to determine the root cause. We demonstrate that, by implementation, the methodology provides a feasible solution to the diagnostic problem.

Acknowledgements

I would like to offer my sincerest gratitude and appreciation to my supervisor, Dr. Thomas Maibaum, who has supported me throughout my thesis with his patience and knowledge and provided me with invaluable guidance and suggestions. Without him this thesis would not have been completed or written.

I would like to thank Dr. George Karakostas and Dr. Alan Wassyng for being on my thesis committee and for their valuable comments and corrections on my thesis.

Thanks also to Systemware Innovation Corporation (SWI), for suggesting the topic and permission to use information from work developed at SWI. In particular, I would like to thank David Tremaine, Jim Picha and Jeff McDougall from SWI, for their guidance and comments on my implementation work.

I am deeply grateful to Dr. Thomas Maibaum and Dr. Chris George for their help in carefully proofreading this thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Complexity and reliability are like two sides of a coin for a software system. On one side, users expect the system to be sophisticated enough to provide comprehensive functions and accomplish the majority of tasks: this is the main reason that software systems have grown visibly in size with complex architecture and numerous lines of source code. On the other side, users expect the system to be reliable enough to accomplish tasks accurately and securely. Unfortunately, as many applications require high reliability and availability, software becomes one of two major contributors to system failure, the other being system administration [Gra85]. Yuan *et al.* [YMX$^+$10] have concluded that, as software systems have grown in size and complexity, it has become increasingly difficult to deliver bug-free software to end users. Most system failures that occurred during runtime of production, including crashes, hangs, incorrect results and other software anomalies, are directly caused by system defects; as a result, diagnosis of software defects becomes an important task in software development and maintenance.

When a system failure occurs during production runtime, regardless of the root cause, support engineers are usually called upon to investigate the problem and come up with a feasible solution within a tight time frame. The conventional diagnosis of software defects requires support engineers to study and understand what has happened to the system when failure occurs. In order to narrow down the root cause, support engineers normally start investigation by reproducing the failure in the system. In reality, as Yuan *et al.* [YMX$^+$10] have revealed, many circumstances make such failure reproduction impossible or forbiddingly expensive. Customer's privacy concerns can make failure reproduction infeasible. For example, companies are prohibited from releasing their databases to software vendors for troubleshooting purposes. Furthermore, it is difficult to have the exact same execution environment, which includes hardware, software, network, third-party applications and so on; as a result, diagnosis of software defects is a challenging task in terms of time and effort.

Software developers and support engineers have turned to external resources at their disposal, such as third-party tools, to help with diagnosis of software defects. There are also internal resources of information available within the software system, i.e., system logs. Xu *et al.* [XHF$^+$09b] described system logs as providing detailed information that reflects the original developers' ideas about noteworthy or unusual events. A system log could be generated by various techniques from a simple *print* statement to a complex logging and monitoring framework, such as Apache log4j. Software developers have used system logs to record program variable values, trace execution, report runtime statistics, and even print out full-sentence messages designed to be read by end users [XHF+09b]. This makes the systems log a very helpful resource for diagnosis of software defects.

Since systems logs contain useful but large amounts of runtime information of a software system, on the occasion of a system failure, the customer sends the system logs to the vendor of the software system. Instead of the conventional failure-reproduction approach, support engineers at the software vendor have to manually examine runtime information recorded in the customer's system logs, applying their expertise of the software system, in order to narrow down the possible root cause, and work out a feasible solution. However, as large software systems are complex, they may generate thousands of logs in a relatively short time frame for one system failure. Analyzing such a large amount of information not only requires expert knowledge, but also turns into it an extremely time-consuming task [YMX+10].

Automated techniques are needed to improve the efficiency and quality of such an analysis process. There exists research work ranging from searching and exploring interrelations between log entries and source code, to diagnosing detected problems by analyzing runtime log files [XHF+09b]. The methodology presented in this thesis has combined elements of source code analysis, log analysis and data mining techniques, to provide a feasible solution to automatic diagnosis of software defects by making use of system logs.

Systemware Innovation Corporation (SWI), a software services and solutions company, maintains a legacy software system that was developed years ago. The software system provides online services to end users in the financial industry. Since there has not been much development work done on the system, it is mainly maintained by support engineers, who are also responsible for responding to end users' requests when system failure occurs. As the original programmers who developed this software system are no longer available, the system totally relies on support engineers' expertise to solve all potential problems. The system generates logs to record its runtime information. These are the major resource that support engineers have access to when investigating end users' problems. They follow the process of conventional log analysis; as a result, they have experienced inefficiency in daily practice.

SWI seeks a different approach to improve the overall procedure. They expect that the alternative approach will be able to minimize manual involvement in the process of log analysis. More specifically, the alternative approach will not have to so rely so heavily on support engineers' expertise to solve end users' problems, and so reduce their workload by a considerable amount. The alternative approach will feature certain automation techniques to achieve those improvements.

SWI contacted the McMaster Centre for Software Certification at McMaster University and expressed their concerns and requests. We conducted extensive research in both industrial and academic domains and proposed a feasible solution to address their concerns. In order to demonstrate the effectiveness of our solution, we have partnered with SWI to implement a research project named the Log Analysis Project in an Automatic Approach.

1.2 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 presents an overview of the problem that we deal with in this thesis. This chapter provides more background information about the challenges encountered in software development and maintenance. It brings up the concept of log analysis in a conventional approach, and explains the limitations and difficulties of conventional log analysis as it is applied in industrial practice. Finally, it presents a brief description of our research project, which is implemented for the purpose of demonstrating a feasible solution to our partner's problem. The objectives and design details of the project are also presented in this chapter.

Chapter 3 presents several concepts, tools and techniques that have been applied to the design and implementation of our research project. Two abstract data types in computer science, graph and call graph, are introduced in this chapter. Furthermore, abstract notions of an access dependency graph and a program call path are discussed. It also presents the two most common analysis techniques, static analysis and dynamic analysis. A brief comparison between these two techniques provides the rationale behind using static analysis for our project. In addition, data mining techniques, such as sequential pattern mining techniques, are introduced in this chapter. At the end of this chapter, we describe a list of implementation tools that are helpful for the implementation of the project.

Chapter 4 demonstrates related work that have been developed in the fields of program dependency graph, detection and diagnosis of system problems, and sequential pattern mining. It also explains why most of the existing work cannot be directly applied to our specific problem domain.

Chapter 5 presents an overview of modules implemented in our research project. It starts with descriptions of each of the three major components: source code analysis, log analysis, and sequential pattern mining. It explains, in terms of data flows, how these three components are integrated into one application. At the end of this chapter, we describe the preparation of the database for our project and how data are stored and queried during the analysis process.

Chapter 6 presents the implementation details of the first of the three major components, i.e. source code analysis. It starts with an introduction of Java bytecode. It compares the advantages and disadvantages of Java bytecode and plain-text source code and discusses why Java bytecode serves more conveniently than plain-text source code for the purpose of our project. It explains why it is necessary to convert Java bytecode into XML representation, and how it could be done with the help of a third-party application. It continues by explaining how the converted Java bytecode can be used in two important analysis processes, which are finding entities, inheritance relations among entities and finding entities responsible for system logging. It provides corresponding definitions and explanations, along with implementation details of each analysis process. It clarifies several concepts of the access dependency graph that are used consistently in this thesis, before describing the overall process of building the access dependency relations. In this description, it explains the process of parsing instructions of calling methods and describes the approach to resolving the dynamic binding issue. Finally, it presents the techniques that are used to store dependency relations in memory during analysis and into the database afterwards.

Chapter 7 presents the implementations details of the second of the three major components, i.e., log analysis. It starts with the description of parsing logs in log files. The process also includes extracting necessary information from logs and matching each log with the corresponding logging point in source code. It describes how to utilize the access dependency relations to build up the access dependency graph. It also introduces two graph traversal algorithms: breadth-first search and depth-first search. It also describes the process of searching for sequential patterns in historical logs and analyzing these patterns to form a knowledge base of the system activities in the past. The most important subject matter in this chapter is the general process of log analysis. More specifically, the process of analyzing a sequence of logs to reveal the corresponding program executions at runtime. The process includes two different approaches: matching the sequence of logs with program call paths and matching the sequence of logs with existing sequential patterns. Through the description of these two approaches, it demonstrates that the methodology and system design of our project is capable of providing a feasible solution to the problem of diagnosis of software defects.

Chapter 8 presents the last of the three major components, i.e., sequential pattern mining. It describes the rationale behind the decision to apply the sequential pattern mining technique in the log analysis process. It seeks to demonstrate that, by mining potential sequential patterns hidden in the log analysis results, this technique can provide an extended insight of the software system from a statistical point of view. It presents an example of mining sequential patterns from some sample data. It describes the output of running an implementation of the mining technique and discusses the implications of these findings for our log analysis project.

Chapter 9 presents an evaluation of the implementation of the log analysis project. Experiments are conducted by running the implementation on a partner's legacy system and analyzing sample logs generated by the system. The experimental results are collected and verified manually, in order to demonstrate the effectiveness of our implementation in both log analysis and sequential pattern mining aspects.

Chapter 10 is the last chapter of this thesis. It describes the contribution of the log analysis project presented in this thesis. It discusses certain limitations with the current implementation and provides suggestions on the potential future work that may be conducted in the future for the purpose of improvement.

Chapter 2

Problem Definition

In this chapter, we present an overview of the problem that we deal with in this thesis. This chapter provides a background to the challenges encountered with diagnosis of software defects in software development and maintenance. We introduce the concept of log analysis in a conventional approach, explain the limitations and difficulties as the conventional log analysis is applied in industrial practice. Finally, we present a brief description of our research project, which is implemented for the purpose of demonstrating a feasible solution to our partner's problem.

2.1 Background

Moore's law, which is also known as "18 months" law, predicts a long-term trend in the history of computing hardware. It states that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every 18 months. The phenomenon of the evolution of computer technologies has sufficiently corroborated Mr Moore's foresight. With the support of enhanced computer hardware, software systems tend to have complex architecture, sophisticated functions, while maintaining necessary efficiency, accuracy and reliability; however, it is unavoidable that, with growing complexity, software defects exist in almost every single large software system. They have caused concern to software practitioners due to the potential cost in terms of financial expense and human effort. Computer scientists and software engineers have dedicated tremendous effort to the search of feasible techniques for diagnosis of software defects. Conventional log analysis works well in certain less complex software systems, but encounters difficulties when deployed in larger complex software systems.

2.2 Conventional Log Analysis

We present several definitions of a log and its related concepts before describing details of conventional log analysis.

2.2.1 Logs and Related Concepts

Logs and Log Files

A log is considered to be a record of the activities of a system and is often stored in a file, which is called a log file. A log file contains a large number of entries that record runtime information of a system. Log entries in a log file are in time sequence. Contents of logs are in plain-text form. When certain log contents cannot be interpreted directly, a log analysis tool is required to help with translation and interpretation. In general, logs are helpful resources for software defect detection and diagnosis.

System Logs

Different software systems have different conventions for recording logs. A system log is one of the most common logs in software systems. The system log file contains events that are logged by the operating system components. These events are often predetermined by the operating system itself. In summary, a system log reveals data and status of the operating system during program executions, such as processor speed, memory usage, system calls and so on.

Transaction Logs

A transaction log is a data collection that captures the type, content and time of transactions made by a person from a terminal with that system [RB83]. The users of transaction logs may be humans or computer programs acting on behalf of humans. Interactions are the communication exchanges that occur between users and the system. Either users or the system may initiate elements of these exchanges [Jan06]. A transactions log is often seen in database systems and web servers. Most transaction logs are server-side recordings of interactions. Both system logs and transaction logs follow certain standard formats. For example, system log formats are predefined during system design and well documented, and typical transaction log formats follow the standard defined by the World Wide Web Consortium (W3C).

Application Logs

An application log has the most diversified log formats, in contrast to the log formats of system logs and transaction logs. Such diversity relates to the fact that applications are highly customized. The implication is that logs are generated by output statements that developers insert into source code, for the purpose of debugging and troubleshooting executions of their application; therefore the syntax and semantics of log contents are application-specific or even developer-specific. The diversity of application log formats leads to the complication of log analysis in the process of diagnosing of software defects.

2.2.2 What is Log Analysis?

Log analysis is a process of identifying an anomaly among log messages that indicates a potential problem with the software system, extracting recorded runtime information of program execution to perform the investigation, applying human expertise to figure out the root cause of the problem, and eventually working out a feasible solution to the problem. We break down the whole process of log analysis into the following steps:

1. Anomaly Detection

   Since unusual log messages often indicate the source of the problem, it is natural to formalize log analysis as an anomaly detection problem in machine learning [YMX+10]. Even though not every single log message directly presents the root cause of the problem, in general, log messages are still an *indicator* to help software end users become aware of potential problems that occur during system runtime, such as system crash, execution failure, data storage problem, improper configuration and so on. This can be done by manually going over lines of logs to identify the unusual log messages, but there are also assistant tools available to help users locate the suspect log messages more quickly and accurately. Such tools parse logs as input and match characteristic strings, i.e., status code, to capture those log messages indicating anomalies.

2. Information Extraction

   Depending on the log format, a log message contains multiple fields of information that reflect the runtime information of program execution, such as date and time, status code, input and output of program, dynamic values of program variables, debugging messages and so on. Once unusual log messages are identified, such information is extracted and formalized, so that relevant information can be connected and interpreted during problem analysis.

3. Problem Analysis

   After collecting all necessary information from unusual log messages, support engineers apply their knowledge and experience of the software system in order to connect runtime information to the source code of the software system. Specifically, they may want to know which parts of the source code, i.e., methods, have been executed and resulted in the log messages identified. They may be interested in the order of program execution, i.e., execution paths of methods, which are reflected in the specific sequence of log messages. These are all pretty valuable analysis results that not only reveal the root cause of the problem, but also essentially help support engineers work out a feasible solution to the problem.

2.2.3 Difficulties with Conventional Log Analysis Revisited

The log analysis process previously presented is typical and conventional, but not efficient. In this section, we revisit factors attributed to the inefficiency of conventional log analysis with more detailed discussions.

Lack of Experience

From the description of conventional log analysis, it's not hard to realize that it involves manual participation. Even though logs contain various information revealing system runtime operations, it is far from sufficient for support engineers to directly conclude the root cause of a potential problem. It still requires support engineers to possess sufficient knowledge and experience of the software system, so that they can combine their expertise with information acquired from logs to successfully solve the problem. Knowledge and experience are always valuable, but not easily attainable. Support engineers with insufficient knowledge or experience may encounter difficulty in correctly identifying the root cause, and as a result, implement an improper solution.

Lack of Precise Runtime Information

In industry, the common practise suggests that, in case of system failure, logs generated by the software system are the only data source provided to support engineers by the software end user. Conventional log analysis implies that support engineers have to manually examine these logs to extract useful information about the problem. As

Yuan *et al.* [YMX+10] have observed, many logs contain hundreds or even thousands of messages, and each message can provide some information about what could have happened in the failed production run. Not all possible clues are relevant. It requires a non-trivial, lengthy analysis to filter out irrelevant information. It is a time and effort consuming task for most support engineers.

Lack of Time for Diagnosis

However, time is one of the things that software end users cannot afford to lose in cases of system failure, which could be in the form of a system crash or complete shutdown. For example, one of the essential requirements for financial software is availability, which means the software system must be available 24/7. Any system downtime could cause considerable financial loss. That is the reason why in a case of system failure, the log analysis needs to be effective in quickly zooming into the right root cause within only a few rounds of interaction with software end users [YMX+10]. Conventional log analysis with intense manual involvements may be insufficient to meet such a stringent requirement.

Considering such difficulties with conventional log analysis, we have turned our focus to searching for a different approach to log analysis, in order to improve the overall efficiency and accuracy.

2.3 The Log Analysis Project

In this section we provide a brief overview of the log analysis project. The implementation work presented in this thesis is the major part of the project. The system modules designed in the project are presented abstractly. Details of implementation are described in following chapters.

The Methodology

After conducting research on the current state of the art in both industrial and academic domains, we have proposed a methodology which combines source code analysis, log analysis and data mining techniques, to detect anomalies among plain-text logs, diagnose software defects, and narrow down the range of source code lines to help determine the root cause of detected problems. The log analysis project in an automatic approach is built upon this methodology, in order to demonstrate a feasible solution to our partner's problem.

The Objectives

To be general and practical, we have four design objectives:

- **Unnecessary re-execution of the program:** For practicality, the methodology only assumes the availability of the source code of the target program and logs generated from a failed execution. It is completely unnecessary to re-execute the program in order to re-generate the output of a failed execution.
- **Capability of error diagnosis:** The methodology is capable of generating a report on error detection, including specifying source code lines that generate error logs, and modules or methods that potentially cause the error.
- **Accuracy:** Information reported by the methodology needs to be accurate. It avoids leading a support engineer to making an incorrect investigation, wasting his effort and stalling the whole diagnostic process.

- **Precision:** Information reported by the methodology needs to be precise. Too many possibilities of error diagnosis will not help a support engineer narrow down the root cause of a failed execution. The methodology applies data mining techniques, i.e., sequential pattern mining techniques, to refine analysis results, in order to provide more precise and more specific results.

The Functionalities

The log analysis project does most detailed analysis in a static manner. The source code of the software system and the logs generated during production runtime are two main resources it requires. The log analysis project provides several functionalities:

1. Parse source code and analyze dependency relations between classes and methods.
2. Parse source code and analyze methods that potentially generate logs during program execution.
3. Parse logs and match each log with the corresponding method.
4. Identify potential system problems recorded in logs.
5. Determine the program execution path that is reflected by the investigated sequence of logs.
6. Recognize sequential patterns among logs that reveal interrelations among system problems in the past.
7. Conclude analysis results and provide suggestions to support engineers.

The System Modules

The implementation of the log analysis project is broken down into multiple system modules corresponding to the functionalities above. The system modules are:

- **Source Code Parser**: Parse source code and identify classes and methods as entities in source code.
- **Source Code Analyzer**: Analyze dependency relations between classes and methods.
- **Log Parser**: Parse logs and match each log with the corresponding class and method.
- **Log Analyzer**: Identify potential system problems recorded in logs. Determine program execution path that is indicated by the investigated sequence of logs. Conclude analysis results and provide suggestions to support engineers.
- **Sequential Pattern Mining Tool**: Recognize sequential patterns among logs that reveal interrelations between system problems.
- **Module Controller**: Control inputs and outputs of system modules.

2.4 Summary

In this chapter, we have discussed the background of our problem, from the general point of view to our partner's specific point of view. We have also proposed our solution to the problem and provided a brief overview of the project that this thesis is built upon.

In the next chapter, we will present several concepts, tools and techniques that have been applied to the design and implementation of our project.

Chapter 3

Tools and Techniques

As sophisticated software systems have complex architectures and large number of lines of source code, there has been increasing interest in tools and techniques that are capable of performing sophisticated program analysis. In this chapter, we will present common analysis tools and techniques for program learning and data mining. We will also discuss how these tools and techniques are applied in our project implementation.

3.1 Graphs

Graphs are appropriate models for depicting many problems that arise in the fields of computer science and software engineering. Control Flow Graph (CFG), Data Flow Graph (DFG), Component Graph (CG) etc. allow the user to take an analytical approach to understanding and characterizing software architecture, static and dynamic structure and meaning of the programs [Cho05]. It is more convenient that the structure of source code should be visualized with a pictorial sketch by graphs. That is why graphs are always preferred by software engineers and researchers to demonstrate design, study system architecture and analyze source code.

Corresponding to various graphical models, there are a number of graph analysis techniques available for software applications. In Control Flow Analysis, a control flow graph is used to analyze and understand how control of the program is transferred from one point of the program to another. Similarly, Data Flow Analysis uses a data flow graph to show and analyze the data dependencies among the instructions of the program. In Component Graph Analysis, a component graph identifies the components of a program, shows the use relations among those components and is very useful in software architecture identification and recovery [Cho05]. Call Graph Analysis utilizes a call graph to detect dependency relations and calling sequences among entities of the program.

For the purpose of our log analysis project, we have also adopted the concept of an Access Dependency Graph from the research work by Iqbal [Iqb11], which is an extension of a call graph. The analysis technique, Access Dependency Analysis, is based on the concept of an access dependency graph. The access dependency analysis utilizes a graphical model to reveal the dependency relations among entities in the Java environment.

3.1.1 Call Graph

Definition

Fenton and Pfleeeger [FP98] define a *call graph* as a directed graph representing the calling relation between a program's modules. It has a specific root node, corresponding to the highest-level module and representing an abstraction of the whole system.

In mathematics, a call graph is a directed graph $G = (N, E)$ with a set of nodes N and a set of edges $E \subseteq N \times N$. A node $u \in N$ represents a program procedure and an edge $(u, v) \in E$ indicates that procedure u calls procedure v [Iqb11]. As an example, consider the call graph in **Figure 3.1**. It has a set of nodes $N = \{a, b, c, d, e, f\}$ and a set of edges $E = \{(a, b), (a, c), (b, d), (b, e), (c, e), (c, f)\}$.

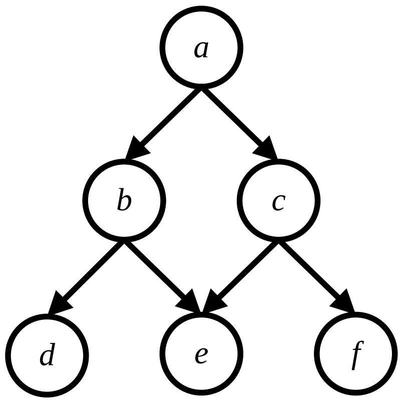


Figure 3.1: Call Graph

The nodes of a call graph represent the methods being either callees or callers; their edges represent the calling relations between the methods. Since call graphs are directed graphs, every edge has an explicit source and target node, representing the calling and called procedure, respectively. Cycles in a call graph represent recursion [Kas04].

There are some concepts related to call graphs that we would like to clarify, before moving on to the description of an access dependency graph.

Call Graphs in a Java Context

For two classes A and B, if A's method $a()$ calls B's method $b_1()$, $b_2()$, ..., $b_n()$, then we consider each of these entities, $A : a()$, $B : b_1()$, $B : b_2()$, ..., $B : b_n()$, as nodes of a call graph and with the following caller-callee relations:

$A : a() \rightarrow B : b_1()$

$A : a() \rightarrow B : b_2()$

$\vdots$

$A : a() \rightarrow B : b_n()$

where the entity on the left of the arrow is the caller (node) and the entity on the right of the arrow is the callee (node).

Static Call Graphs vs. Dynamic Call Graphs

Call graphs can be static or dynamic. A dynamic call graph is a record of an execution of the program; therefore a dynamic call graph can be exact for only one execution of the program, but there can be different dynamic call graphs for the same program when it is executed multiple times.

A static call graph is a call graph intended to represent every possible execution of the program; therefore a dynamic call graph is a sub-call graph of a static call graph for the same program. The exact static call graph is *undecidable*, which means a static call graph is generally an *approximation* of program executions. A static call graph could possibly include some call relations that would never occur during actual program executions.

Dynamic Call Graphs vs. Program Call Paths

As previously stated, a dynamic call graph records only one execution of the program, which reflects the calling sequence of entities in the program during that execution. Since a dynamic call graph is a directed graph, it visualizes one or multiple paths from the starting node toward the ending node; therefore a dynamic call graph is a set of program call paths, where each program call path is a unique sub-call graph of the dynamic call graph from one node to another node. When there is one and only one path from the starting node to the ending node in a dynamic call graph, this dynamic call graph is also a program call path.

3.1.2 Access Dependency Graphs

The simple notion of a call graph works well in traditional non-object-oriented programming languages like C, whose programs involve only explicit method calls. However, in an object-oriented programming language like Java, where methods are encapsulated inside classes, the relations among entities are far more complicated than a simple call graph could represent. Java language features complex programming processes such as inheritance and dynamic binding, which can introduce implicit dependency relations among methods that are not explicitly present in the source code. Since Java programs involve both explicit and implicit method calls, the entity relations cannot be conveniently represented by a simple call graph.

For example, there are three classes A, B and C. Class B extends class A and overrides a method $m()$ in class A. There is a method $n()$ inside class C. Consider the scenario of an explicit call from class C's method $n()$ to class A's method $m()$. This is a typical dynamic binding issue in an object-oriented language. The call might actually result in a call to class B's method $m()$ instead of the one in class A.

Definition

Iqbal [Iqb11] has proposed a notion called an *Access Dependency Graph* in his thesis:

> An access dependency graph is a directed graph $G = (N_m, N_f, E)$ with a set of method nodes N_m, a set of field nodes N_f and a set of edges $E \subseteq (N_m \cup N_f) \times (N_m \cup N_f)$. A node $m \in N_m$ indicates a method node and is of the form *ClassName*:*MethodName*. A node $f \in N_f$ indicates a field node and is of the form *ClassName*:*FieldName*. An edge $(m, e) \in E$ (where $m \in N_m$) may indicate one of the following two kinds of dependency:
>
> - An explicit method call from method m to method e if $e \in N_m$, or an explicit access of a field e from method m if $e \in N_f$.
> - An implicit dependency from method m to method e where an explicit call from somewhere to method m may actually result in a call to method e due to dynamic binding.

Modification

Iqbal's definition of an access dependency graph includes considerations of both methods and fields. For our log analysis project, we are interested in dependency relations between methods; therefore we have modified the original definition as follows:

> An access dependency graph is a directed graph $G = (N_m, E)$ with a set of method nodes N_m and a set of edges $E \subseteq N_m \times N_m$. A node $m \in N_m$ indicates a method node and is of the form *ClassName*:*MethodName*. An edge $(u, v) \in E$ (where $u \in N_m$ and $v \in N_m$) may indicate one of the following two kinds of dependency:
>
> - An explicit method call from method u to method v.

- An implicit dependency from method u to method v where an explicit call from somewhere to method u may actually result in a call to method v due to dynamic binding.

Consider the access dependency graph in **Figure 3.2**. It has a set of method nodes $N_m = \{A : a(), B : b(), C : c(), E : e(), F : f(), X : m(), Y : m()\}$ and a set of edges $E = \{(A : a(), B : b()), (A : a(), C : c()), (B : b(), E : e()), (B : b(), X : m()), (C : c(), F : f()), (C : c(), Y : m()), (X : m(), Y : m())\}$.

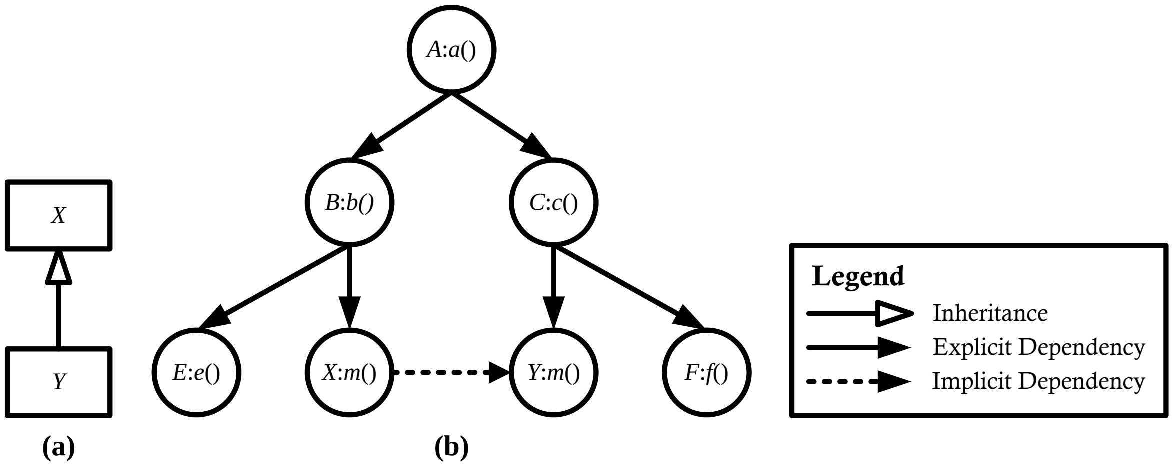


Figure 3.2: (a) Inheritance of Class Y from Class X. (b) Access Dependency Graph.

The first six edges represent explicit dependencies and the last one, which is the edge $X : m() \rightarrow Y : m()$, represents implicit dependency.

3.1.3 Reasons for Choosing Access Dependency Analysis

Access dependency analysis is a notion of analysis technique based on the concept of an access dependency graph. It analyzes dependency relations among entities in the source code environment and presents such relations in the form of an access dependency graph. We opted for access dependency analysis in the log analysis project for the following reasons:

- The source code is the only resource for understanding the structure of our partner's system. The access dependency analysis recognizes dependency relations among entities, i.e., methods, in the source code, and the dependency connections help reveal the structure of the system.
- The core of our partner's legacy system is developed in Java language, which features inheritance and dynamic binding processes that would lead to dependency relations among entities in source code. Access dependency analysis resolves such complex issues and prepares dependency information for the eventual log analysis process.
- The result of access dependency analysis is recorded and conveniently maintained. It can be used repeatedly for the log analysis, as long as the source code remains intact. When there are changes to the source code, in order to obtain updates on dependency relations among entities, we only need to perform the access dependency analysis once again.

3.2 Program Analysis Techniques

In computer science, program analysis is the process of studying and analyzing potential behaviors of computer systems. Static program analysis and dynamic program analysis are two main techniques. Traditionally, static program analysis and dynamic program analysis have been viewed as two separate domains. As computer technology and technique have evolved and integrated, practitioners and researchers consider static program analysis and dynamic program analysis as complementary approaches to certain software analysis matters, such as program comprehension, system testing, software verification and so on. Static analysis provides global information concerning the program structure, whereas dynamic analysis investigates the runtime behavior of the program [Fai78].

3.2.1 Static Program Analysis

Static program analysis is performed without executing the program. It examines program source code and reasons over all possible behaviors that might arise at runtime. Compiler optimizations are standard static program analysis [Ern03].

Static program analysis operates by building a model of the program state, then determining how the program reacts to this state. Because there are many possible executions, the analysis must keep track of multiple different possible states. The process results in many possible program execution paths. It is not reasonable to consider every possible runtime state of the program; instead, static program analysis uses an abstracted model of the program state. As a result, the analysis output may be less precise [Ern03].

Because static program analysis is concerned with analyzing the structure of source code, it is particularly useful for discovering logical errors with formal methods and questionable coding practices leading to programming errors [Fai78].

3.2.2 Dynamic Program Analysis

In contrast to static program analysis, dynamic program analysis is performed by executing the program and observing the executions. Testing and profiling are standard dynamic program analysis [Ern03].

Dynamic program analysis examines the actual and exact behavior of program execution. There is little or no uncertainty over what control flow paths were taken, what values were computed, how much memory was consumed, how long the program took to execute, or other quantities of interest; therefore the analysis output is precise [Ern03].

While dynamic program analysis cannot prove that a program satisfies a particular property, it can detect violations of properties as well as provide useful information to programmers about the behavior of their programs [Bal99].

The contrast between dynamic and static program analysis is similar to the contrast between dynamic and static call graphs. Static program analysis derives properties that hold for all possible executions by examining the source code of a program, while dynamic program analysis derives properties that hold for one or more executions by examining the execution of a program [Bal99].

3.2.3 Reasons for Choosing Static Program Analysis

We have presented some advantages and disadvantages of static program analysis and dynamic program analysis. Comparing these two techniques, we opted for static program analysis in the log analysis project for the following reasons:

- There are only two main resources for diagnosis of software defects in our partner's problem: the source code and the logs. Logs are generated from specific instrumented code as an approach to dynamic program analysis, but analyzing logs alone is not sufficient. What is missing is the explicit link to input combinations as in test cases. Unfortunately, the records of user input are not available in this project. Furthermore, dynamic program analysis is usually performed by executing the instrumented program, but running our partner's software system involves their clients' confidential data; as a result, this is not permitted in our project due to privacy and security concerns.

- The disadvantage of dynamic analysis is that its results may not generalize to future executions. There is no guarantee that the test suite over which the program was run (that is, the set of inputs for which execution of the program was observed) characterizes all possible program executions [Ern03]. The incoming data to our partner's legacy system could be in various types and values. It is almost impossible to create test suites covering all possible inputs.

- The diagnosis of software defects focuses only on the one execution that has caused the system failure. Even though information is recorded in the logs, reproduction of exactly the same system failure may be impossible due to the absence of runtime environment and input data. Since static program analysis does not require re-execution of the program, it can perform the necessary analysis with the limited information available.

- Static program analysis can utilize the source code to collect information about the program structure, dependency relations among program modules and all possible program call paths. Such software metrics are connected to logs in order to narrow down the range of source code lines likely to be related to a specific software defect recorded in the logs. This approach provides a feasible solution to our partner's problem given the limited resources of our project.

- There are other software analysis techniques, such as Change Impact Analysis.

Software Change Impact Analysis is defined as the determination of potential effects to a subject system resulting from a proposed software change [Boh02]. The Change Impact Analysis involves both static and dynamic program analysis, but the implication focuses on the scoping changes within the details of a design and the risks associated with changes. It is about the evaluation of many risks associated with the changes, including estimates of the effects on resources, effort and schedule. Since the impact of program design changes is not considered within the scope of our project, strictly speaking, the Change Impact Analysis is not suitable for our purpose.

3.3 Sequential Pattern Mining

Once we have collected a sufficient number of logs, sequential pattern mining could be one helpful technique in analyzing the occurrences of logs, recognizing any possible relationships between logs as patterns, exploring these patterns to discover interrelations between logs and the source code lines that generate them. The sequential pattern mining technique refines the log analysis results, so that they provide sufficient information to support engineers when they investigate runtime system failures.

3.3.1 Definitions

Let $I = \{i_1, i_2, ..., i_n\}$ be a set of all **items**. An **itemset** is a subset of items. A **sequence** is an ordered list of itemsets. A sequence s is denoted by $\langle s_1 s_2 ... s_l \rangle$, where s_j is an itemset. s_j is also called an **element** of the sequence, and denoted as $(x_1 x_2 ... x_m)$, where x_k is an item. For brevity, the brackets are omitted if an element has only one item, i.e., element (x) is written as x. An item can occur at most once in an element of a sequence, but can occur multiple times in different elements of a sequence. The number of instances of items in a sequence is called the **length** of the sequence. A sequence with length l is called an **l-sequence**. A sequence $\alpha = \langle a_1 a_2 ... a_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle b_1 b_2 ... b_n \rangle$ and β is a **super-sequence** of α, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < ... < j_n \leq m$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}, ..., a_n \subseteq b_{j_n}$ [HPY05].

A **sequence database** S is a set of tuples $\langle sid, s \rangle$, where sid is a **sequence id** and s is a sequence. A tuple $\langle sid, s \rangle$ is said to *contain* a sequence α, if α is a sub-sequence of s. The support of a sequence α in a sequence database S is the number of tuples in the database containing α, i.e., $support_S(\alpha) = |\{\langle sid, s \rangle | (\langle sid, s \rangle \in S) \wedge (\alpha \sqsubseteq s)\}|$. It can be denoted as $support(\alpha)$ if the sequence database is clear from the context. Given a positive integer *min_support* as the **support threshold**, a sequence α is called a **sequential pattern** in sequence database S if $support_S(\alpha) \geq min\_support$. A sequential pattern with length l is called an **l-pattern** [HPY05].

3.3.2 Methods and Algorithms

The sequential pattern mining problem was first introduced by Srikant and Agrawal [SA96], and since then the goal of sequential pattern mining is to discover all frequent sequences of itemsets in a dataset. Recent studies have developed two major classes of sequential pattern mining methods: (1) a *candidate generation-and-test* approach, represented by (i) **GSP**, a horizontal format-based sequential pattern mining method, and (ii) **SPADE**, a vertical format-based method; and (2) a *sequential pattern growth* method, represented by **FreeSpan** and **PrefixSpan** and their further extensions, such as **CloSpan** for mining closed sequential patterns [HPY05].

GSP

GSP (**G**eneralized **S**equential **P**atterns) is a horizontal data format-based sequential pattern mining developed by Srikant and Agrawal [SA96] by extension of their frequent itemset mining algorithm, Apriori.

GSP adopts a multiple-pass, candidate-generation-and-test approach in sequential pattern mining. The algorithm makes multiple passes over the data. The first pass determines the support of each item, that is, the number of data-sequences that include the item. At the end of the first pass, the algorithm knows which items are frequent. Each of such items yields a 1-element frequent sequence consisting of that item. Each subsequent pass starts with a seed set: the frequent sequences found in the previous pass. The seed set is used to generate new potentially frequent sequences, called candidate sequences. Each candidate sequence has one more item than a seed sequence; so all the candidate sequences in a pass will have the same number of items. The support for these candidate sequences is found during the pass over the data. At the end of the pass, the algorithm determines which of the candidate sequences are actually frequent. These frequent candidates become the seed for the next pass. The algorithm terminates when there are no frequent sequences at the end of a pass, or when there are no candidate sequences generated [SA96].

The bottleneck of an Apriori-based sequential pattern mining method comes from its step-wise candidate sequence generation and test. GSP, though it benefits from the Apriori pruning, still generates a large number of candidates [HPY05].

SPADE

SPADE (**S**equential **PA**ttern **D**iscovery using **E**quivalent classes) is an Apriori-based vertical data format sequential pattern mining algorithm [Zak01]. The Apriori-based sequential pattern mining can also be explored by mapping a sequence database into the vertical data format which takes each item as the center of observation and takes its associated sequence and event identifiers as data sets. To find a sequence of length-2 items, one just needs to join two single items if they are frequent and they share the same sequence identifier and their event identifiers (which are essentially relative timestamps) follow the sequential ordering. Similarly, one can grow the length of itemsets from length two to length three and so on. This forms the SPADE algorithm [HPY05].

The SPADE algorithm may reduce the access of sequence databases since the information required to construct longer sequences is localized to the related items and/or subsequences represented by their associated sequence and event identifiers. However, the basic search methodology of SPADE is similar to GSP, exploring both breadth-first search and Apriori pruning. It has to generate a large set of candidates in breadth-first manner in order to grow longer subsequences. Thus most of the difficulties suffered in the GSP algorithm will reoccur in SPADE as well [HPY05].

FreeSpan

FreeSpan (**Fre**qu**e**nt pattern-projected **S**equential **pa**tter**n** mining) is a sequential pattern growth method developed by Han *et al.* [HPMA^{+}00]. Its general idea is to use frequent items to recursively project sequence databases into a set of smaller projected databases and grow subsequence fragments in each projected database. This process partitions both the data and the set of frequent patterns to be tested, and confines each test being conducted to the corresponding smaller projected database.

The performance study shows that FreeSpan mines the complete set of patterns and is efficient and runs considerably faster than the Apriori-based GSP algorithm [PHMA+01]. However, as Pei *et al.* point out, the projected databases in FreeSpan have to keep the whole sequence in the original database without length reduction, which could potentially affect the efficiency of memory usage.

PrefixSpan

PrefixSpan (**Prefix**-projected **S**equential **pa**tter**n** mining), proposed by Pei *et al.* [PHMA+01], provides a more efficient processing compared to other sequential pattern mining methods. The major idea of PrefixSpan is that, instead of projecting sequence databases by considering all the possible occurrences of frequent subsequences, the projection is based only on frequent prefixes because any frequent subsequence can always be found by growing a frequent prefix. According to the performance study conducted by Pei *et al.*, PrefixSpan outperforms both the GSP algorithm and the FreeSpan method in mining large sequence databases; therefore we decided to adopt the PrefixSpan algorithm in our log analysis project.

CloSpan

CloSpan (**Clo**sed **S**equential **pa**tter**n** mining) is developed by Yan *et al.* [YHA03] as an extension of PrefixSpan. While the sequential pattern mining algorithms developed so far have good performance in databases consisting of short frequent sequences, when mining long frequent sequences, or when using very low support thresholds, the performance of such algorithms often degrades dramatically.

Yan *et al.* [YHA03] proposed an alternative but equally powerful solution: instead of mining the complete set of frequent subsequences, one mines frequent *closed subsequences* only, i.e., those containing no super-sequence with the same support. CloSpan is developed to mine these patterns. It can produce a significantly less number of sequences than the traditional (i.e., full-set) methods while preserving the same expressive power, since the whole set of frequent subsequences, together with their supports, can be derived easily from our mining results.

3.4 Implementation Tools

We decided to apply different tools in the implementation of log analysis project, including a development tool, a dependency analysis tool and a data repository tool. We have successfully integrated such tools to realize the objectives and functionalities of the project.

Java Environment

We use Java programming language to implement the log analysis project for the following reasons:

- Java is a reasonably structured and comprehensively developed object-oriented programming language. It provides extensive functionalities to cope with many different software engineering issues during software development and maintenance.

- Java has good open source project support, detailed documentation, large numbers of sample implementations on the web.

- Java is platform independent. A Java application that runs on one platform does not need to be recompiled to run on another; as a result, the log analysis project can be deployed on various mainstream operating systems without implementing platform specific versions.

- Our partner's legacy system is implemented using Java. In order to analyze the source code in the form of Java bytecode (*class* file), we need specific tools to interpret contents in Java bytecode. There are existing bytecode analysis tools that export external functions to projects developed in a Java environment.

Dependency Finder

Dependency Finder [Tes12a, Tes12b] is a suite of tools for analyzing compiled Java code. The core is a powerful dependency analysis application that extracts dependency graphs and mines them for useful information. This application comes in many forms for ease of use, including command-line tools, a Swing-based application, a web application ready to be deployed in an application server, and a set of Ant [Fou12a] tasks.

Among the suite of tools available with Dependency Finder, the specific tools that are suitable for our purpose are *ClassReader* and *DependencyExtractor*. Iqbal [Iqb11] has compared these tools experimentally and concluded that both of these tools generate XML files that are ready for parsing and interpretation. *ClassReader*'s XML is a one to one XML representation of a *class* file, while *DependencyExtractor*'s XML specifically encompasses the dependency information. However, although *DependencyExtractor* is an excellent tool for extracting a dependency relationship from *class* (also *jar* and *zip*) files, that dependency information does not incorporate dynamic binding properly. We have already discussed the importance of dynamic binding issue in access dependency analysis, so for this particular reason, we have decided to use *ClassReader*'s XML for the access dependency analysis in our project.

XML

Having an XML representation of a Java *class* file gives us certain advantages over having other kinds of representations, such as Java bytecode and plain-text Java source code. First, XML is well formatted so that, when it is necessary, support engineers can manually interpret its contents without much difficulty. Second, due to the specific format, it is convenient to program an application for automatic interpretation of XML, as is the case in our log analysis project. Finally, XML is used not only for representation, but also as a data repository of useful information. The majority of extensively used programming languages like Java have developed versatile built-in facilities to parse and manipulate XML.

The Database

Databases are commonly used as a data repository in many software applications as well. The term database normally implies an organized collection of data. Traditional databases are organized by records and files. Nowadays, most databases are organized in digital form. The structure of a database system includes the database with data collection and the database management system (DBMS). The DBMS enables database administrators to store, modify and retrieve information from a database. In the log analysis project, we use a database along with a DBMS to store a potentially large number of archive logs generated by the software system in the production environment. Having the database as a central repository of logs allows convenient yet secure access, locally and remotely, by end users and support engineers, respectively. We can execute handy SQL queries on the database to store or fetch necessary information. Furthermore, logs in the database can be mined with sequential pattern mining techniques in the search for hidden sequential patterns among logs. Besides archive logs, we also use the database to store analysis results as records for future reference. We decide to use the MySQL database management system [Cor12]. Even though other standard database systems, such as Oracle and SQL Server, are also suitable, we opt for MySQL because it is free software and features the functionality necessary for the purposes of our project.

3.5 Summary

In this chapter, we have presented certain theoretical concepts and definitions, analysis tools and data mining techniques that are applied in our log analysis project.

In the next chapter, we will present some of the related work that has been researched and developed in the fields of program dependency analysis, detection and diagnosis of system problems, and sequential pattern mining. The chapter also explains why some of this existing work cannot be directly applied to our specific problem domain.

Chapter 4

Related Work

There has been extensive research and development work in the fields of program dependency analysis, detection and diagnosis of system problems, and sequential pattern mining. In this chapter, we present some of the related work and discuss why this existing work cannot be directly applied to our project domain.

4.1 Work Related to Program Dependency Analysis

Much related work has been performed over the decades in the area of program dependency analysis and representations.

Ryder [Ryd79] introduced the notion of a call graph as an acyclic graph to collapse the dynamic relations between procedures in a program. Ryder also proposed an algorithm to compute the call graph for languages that permit procedure parameters but disallow recursion. Callahan *et al.* [CCHK90] extended Ryder's work to support recursion. However, this work was not carried out in object-oriented domains, so they did not face the challenges that arise in object-oriented content.

More extensive research was carried out in the area of program dependency representations. Ottenstein *et al.* [OO84] introduced the notion of a program dependency graph (PDG) to facilitate implementation of an internal program representation chosen for a software development environment. Later, Ferrante [FOW87] worked with Ottenstein *et al.* on the extension of this dependence-based representation, to explicitly represent control and data dependences in a sequential procedural program with single procedure. Horwitz *et al.* [HRB90] further extended the PDG to introduce an inter-procedural dependence-based representation, called system dependence graph (SDG), to represent a sequential procedural program with multiple procedures. Although these representations can be used to represent many features of a procedural program, they still lacked the ability to represent object-oriented features in Java software.

Larsen and Harrold [LH96], along with other researchers, extended the SDG for sequential procedural programs to the case of sequential object-oriented programs. Malloy *et al.* [MMKM94] proposed a new dependence-based representation, called the object-oriented program dependency graph (OPDG), for sequential object-oriented programs. Although these representations can be used to represent many features of sequential object-oriented programs, they still lacked the ability to represent some specific features such as interfaces and packages in Java software.

One point worth mentioning is that the dependency graph we intend to generate has to include all necessary information so that log analysis can be carried out successfully, even at the expense of simplicity. For example, we need dependency relations not only among entities in Java source code, but also among internal classes (*class* files) and external classes (*jar* and *zip* files). The extra information is important for the sake of completeness, even though it increases the overall complexity of the dependency graph. On the other hand, as Iqbal [Iqb11] pointed out, there might be mutually recursive or conditionally repeated (i.e. *for* loop) executions: all we need to know in our case are those entities involved in these executions and their dependency relations, while the repetitive call sequences are beyond our scope. So many of the concepts and techniques discussed above, even though brilliant and significant in their specific problem domains, are not quite suitable for our project.

Zhao [Zha98] presented the software dependence graph for Java (JSDG), which extends previous dependence-based representations, to represent various types of program dependence relationships in Java software. The JSDG of Java software consists of a group of dependence graphs which can be used to represent Java methods, classes and their extensions and interactions, interfaces and their extensions, complete programs, and packages respectively. The JSDG can be used as an underlying representation to develop software engineering tools for Java software. Zhao's proposal provides an inspirational approach to representing dependency relations with graphical data structures. Zhao did not provide implementation details of JSDG in his paper, which are essential for practical application development. The paper also missed out details on how to capture entities in a Java environment and recognize dependency relations among various entities. From development point of view, we need more supportive information to actually implement the idea of JSDG into an applicable tool in our log analysis project.

Dependency Finder [Tes12a, Tes12b] has a tool for computing object-oriented software metrics that give users an empirical quality assessment of source code in a Java environment. As a brief recap, Dependency Finder is a suite of tools for analyzing compiled Java code. At its core is a powerful dependency analysis application that extracts dependency graphs and mines them for useful information. This application comes in many forms for ease of use, including command-line tools, a Swing-based application, a web application ready to be deployed in an application server, and a set of Ant [Fou12a] tasks. This application can generate two kinds of XML representations with two separate tools: one is an one-to-one XML representation of a Java bytecode (*class*) file, and the other is an encompassed XML representation of dependency information only. The former representation is suitable for the log analysis project; therefore we decide to integrate the Dependency Finder tool into our implementation.

4.2 Work Related to Detection and Diagnosis of System Problems

Profiling and Trace Analysis

Profiling is a form of dynamic program analysis. It is achieved by program instrumentation. Programmers can instrument either program source code or binary executable form, by inserting instruction codes that monitor specific components in a system. Mytkowicz *et al.* [MCD09] proposed a call-profiling technique to collect minimal information from the running program without negatively affecting the program performance. The call path profiles capture the nested sequence of calls encountered at run-time; thus they are useful for determining which sequences of calls consume the most program execution time and for identifying opportunities for code specialization. They can also reveal any anomaly during execution indicating potential system problems.

Trace analysis analyzes execution traces generated at runtime by instrumented programs to detect or diagnose errors. Liblit *et al.* [LAZJ03] introduced an analysis technique called *automatic bug isolation.* It collects run-time traces from many users to offload the monitoring overhead. The trace information gathered from all executions is analyzed to extract information that is most correlated with the bugs, which helps support engineers detect and diagnose system problems. Liblit later collaborated with Chilimbi *et al.* [CLM$^+$09] in developing a statistical debugging tool called *HOLMES* that isolates bugs by finding paths that correlate with failure. The tool uses iterative, bug-directed profiling to lower execution time and space overheads. The development results indicate that path profiling can help isolate bugs more precisely by providing more information about the context in which bugs occur.

Even though profiling and trace analysis provide dynamic insights of program executions, which substantially improve the efficiency of detection and diagnosis of system problems, there are increasing concerns about the negative impact of program instrumentation. Normally, instruction codes need to be implanted into a target program statically or dynamically to monitor executions, which can slow down the program execution and affect the performance. Moreover, program instrumentation is not always feasible, especially for production software. Software users are understandably reluctant to perform instrumentation on their systems. We decide not to adopt the approach of profiling and trace analysis in our project.

Log Analysis

Most existing log analysis work adopts principles of static program analysis and applies statistical techniques to detecting anomalies indicated by the logs or recurring failures that match known issues.

Lee *et al.* [LIT91] presented a methodology for the analysis of automatically generated event logs from fault tolerant systems. With this methodology, errors are identified based on knowledge of the architectural and operational characteristics of the measured systems. It is found that the number of errors is small, even though the measurement period is relatively long. This reflects the high dependability of the measured systems. The methodology uses multivariate statistical techniques - factor analysis and cluster analysis - to investigate error and failure dependency among different system components. Despite its advantages, this methodology mostly relies on statistical techniques and information collected from event logs, and does not utilize information hidden in source code.

Xu *et al.* [XHF$^+$09a] have conducted extensive studies and research on data mining and machine learning techniques to learn common patterns from a large number of console logs, and to detect abnormal log patterns that violate the common patterns. They proposed a novel application of using data mining and statistical learning methods to automatically monitor and detect abnormal execution traces from console logs in an online system. In this application, they use a two-stage detection system. The first stage uses frequent pattern mining and distribution estimation techniques to capture the dominant patterns (both frequent sequences and time duration). The second stage uses a principal component analysis-based anomaly detection technique to identify actual problems.

However, the error detection and diagnosis from these studies and research are mostly based on learning patterns solely from log messages, without leveraging source code for extracting control-flow and data-flow information; therefore they cannot recreate the execution paths (or partial execution paths) and runtime variable values [YMX$^+$10].

Yuan *et al.* [YMX$^+$10] designed and implemented a practical and effective tool, called *SherLog*, that analyzes logs from a failed production run and source code by leveraging information collected from runtime logs, to diagnose what has happened during the failed production run. This analysis technique requires neither re-execution of the program nor knowledge of log semantics. It automatically infers control-flow and data-flow information that help support engineers investigate the error. The analysis technique proposed by Yuan *et al.* is intelligent and inspirational. It overcomes certain shortcomings of previous methodologies by connecting source code analysis with log analysis, to provide a thorough understanding of any failed execution at production runtime. In contrast to previous methodologies like Xu *et al.*'s, Yuan *et al.*'s methodology does not involve any statistical or data mining techniques in the analysis process.

Conclusion

Considering the fact that large-scale software systems generate a large number of logs at runtime, we should not neglect the significance of statistical and data mining techniques for the detection and diagnosis of system problems. Some techniques, such as sequential pattern mining technique, are able to recognize interrelations between logs and reveal potential system problems. We decide to combine source code analysis, log analysis and sequential pattern mining technique, to implement a more comprehensive solution for detection and diagnosis of system problems.

4.3 Work Related to Sequential Pattern Mining

With the successful development of sequential pattern-growth mining methods, researchers have made great efforts to explore how these methods can be extended to handle more sophisticated mining requests. We will present a few extensions of the sequential pattern-growth mining approach proposed by Han *et al.* [HPY05].

Mining Multi-Dimensional, Multi-Level Sequential Patterns

In many applications, sequences are often associated with different circumstances, and such circumstances form a multiple dimensional space. It is interesting and useful to mine sequential patterns associated with *multi-dimensional information.* Similarly, items in the sequence may also be associated with *different levels of abstraction*, and such multiple abstraction levels will form a multi-level space for sequential pattern mining [HPY05].

There have been numerous studies of mining frequent patterns or associations at multiple levels of abstraction, and mining association or correlations in multiple dimensional space. One of the pattern growth-based methods, such as *PrefixSpan*, can be naturally extended to mining sequential patterns efficiently in such a multi-dimensional, multi-level environment [HPY05].

Constraint-Based Mining of Sequential Patterns

With many sequential pattern mining applications, instead of finding all the possible sequential patterns in a database, a user may often want to enforce certain constraints to find desired patterns. The mining process which incorporates user-specified constraints to reduce search space and derive only the user-interested patterns is called *constraint-based mining* [HPY05].

Constraint-based mining has been studied extensively in frequent pattern mining. In general, constraints can be characterized based on the notion of monotonicity, anti-monotonicity, succinctness, as well as convertible and inconvertible constraints respectively, depending on whether a constraint can be transformed into one of these categories if it does not naturally belong to one of them. This has become a classical framework for constraint-based frequent pattern mining. With the development of the pattern-growth methodology, such a constraint-based mining framework can be extended to the sequential pattern mining process [HPY05].

Mining Top-k Closed Sequential Patterns

Mining closed patterns may significantly reduce the number of patterns generated and is *information lossless* because it can be used to derive the complete set of sequential patterns. However, setting *min_support* is a subtle task: *A too small value may lead to the generation of thousands of patterns, whereas a too big one may lead to no answer found.* To come up with an appropriate *min_support*, one needs prior knowledge about the mining query and the task-specific data, and to be able to estimate beforehand how many patterns will be generated with a particular threshold [HPY05].

Han *et al.* [HPY05] proposed that a desirable solution is to change the task of mining frequent patterns to *mining top-k frequent closed patterns of minimum length min_ℓ*, where k is the number of closed patterns to be mined, top-k refers to the k most frequent patterns, and *min_ℓ* is the minimum length of the closed patterns. They later developed a multi-pass search space traversal algorithm to discover top-k closed sequences. The algorithm finds the most frequent patterns early in the mining process and allows dynamic raising of *min_support* which is then used to prune unpromising branches in the search space.

Mining Approximate Consensus Sequential Patterns

Conventional sequential pattern mining methods may meet inherent difficulties in mining databases with long sequences and noise. They may generate a huge number of short and trivial patterns but fail to find interesting patterns approximately shared by many sequences. In many applications, it is necessary to mine sequential patterns approximately shared by many sequences [HPY05].

Han *et al.* [HPY05] proposed the theme of *approximate sequential pattern mining* roughly defined as *identifying patterns approximately shared by many sequences.* They presented an efficient and effective algorithm to mine consensus patterns from large sequence databases. First, the sequences are clustered by similarity. Then, the consensus patterns are mined directly from each cluster through multiple alignments. A novel structure called weighted sequence is used to compress the alignment result. For each cluster, the longest consensus pattern best representing the cluster is generated from its weighted sequence.

Conclusion

We decide to adopt one of the sequential pattern mining algorithms, i.e., Prefixspan, in our log analysis project. We will monitor experimental results generated by the algorithm and implement applicable extension to refine the results when it is necessary.

4.4 Summary

In this chapter, we have presented much of the related work in the fields of program dependency analysis, detection and diagnosis of system problems, and sequential pattern mining. We have also discussed their applicability issues in our project domain.

In the next chapter, we will present an overview of major components implemented in the log analysis project. We will discuss how these components are systematically integrated into one single application.

Chapter 5

Overview of Project Implementation

In this chapter, we will present an overview of major components implemented in the log analysis project. Three major components, source code analysis, log analysis and sequential pattern mining, are described. It explains how these components are systematically integrated into one application. The utilization of a database as a data repository in our project is also discussed.

5.1 Principles of The Three-Tier Architecture

Client-server architecture is a generic term for any application architecture that divides processing between two or more processes, often on two or more machines [Ree00]. For example, any database application is a client-server application if it handles data storage and retrieval in the database servers, manipulates data outside the database process, and presents data to clients upon request. According to Reese [Ree00], the idea behind the client-server architecture in a database application is to provide multiple users with access to the same data.

Conventionally, the client-server architecture includes two tiers representing processes on the servers and on the clients. As software systems have evolved to complex and sophisticated architecture, more tiers are necessary to logically separate processes. Nowadays, the most commonly used client-server architecture is the three-tier architecture, in which the user interface, functional process logic, computer data storage and retrieval are developed and maintained as independent modules [Eck95]. In general, a three-tier architecture is composed of the following three tiers:

- **Presentation Tier**: This tier consists of a user interface, which communicates with clients by taking clients' requests as input and displaying information and results of services to clients as output.
- **Application Tier**: This tier is also sometimes called the business logic tier. It does most of the data manipulations and necessary data computations. It ensures data flowing between the presentation tier and the data storage tier complies with business rules required by the application end users. In other words, this tier controls most of an application's functionality.
- **Data Tier**: This tier consists of database servers, which provide data storage and retrieval upon request. The database system manages data independently from the application tier, which means any changes to business rules enforced by the application tier should not affect data maintained at this tier.

5.2 Design of Major Components

The design of the log analysis project adopts the principle of client-server architecture. The presentation tier consists of a general Graphical User Interface (GUI) and the data tier consists of a database server. The application tier is broken down into three major components as shown in **Figure 5.1**.

- **Source Code Analysis**: Source Code Parser and Source Code Analyzer
- **Log Analysis**: Log Parser and Log Analyzer
- **Sequential Pattern Mining**: Sequential Pattern Mining Tool

5.2.1 Source Code Analysis

The goal of source code analysis is to collect information about methods that possibly generate logs during program execution, and about the access dependency relations among entities, i.e., methods, in source code. This is achieved by parsing and analyzing source code in the form of Java bytecode (*class* file), which are accomplished by a source code parser and a source code analyzer, respectively. The process of source code analysis is as demonstrated in **Figure 5.2**.

Source Code Parser

The source code parser takes Java bytecode as input, parses its contents to recognize classes and methods, and collects information about each of these entities, such as the signature of a method, the class in which a method is defined, the *class* file name and so on. This is essential information for access dependency analysis and log analysis.

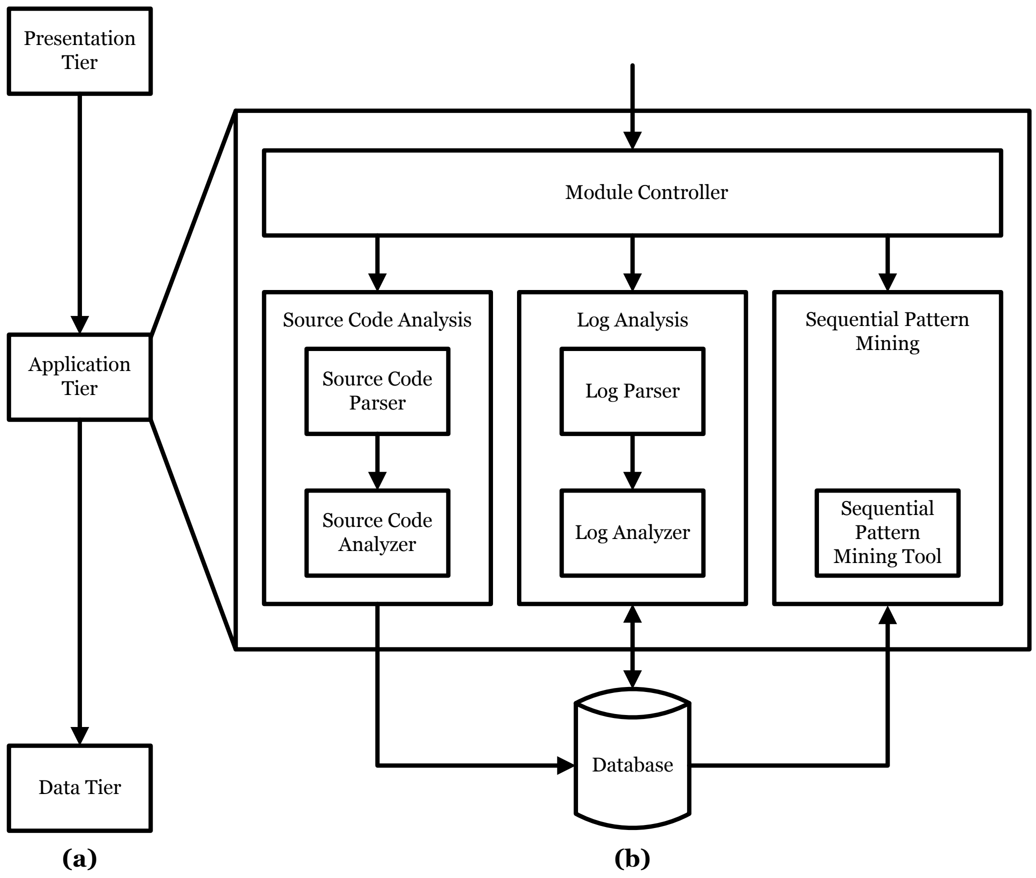


Figure 5.1: (a) Three-Tier Architecture. (b) Major Components of Log Analysis Project.

Java bytecode (*class* file) is the compilation output of Java source code (*java* file). It is a form of program instructions that are executed by a Java virtual machine (JVM). It is not expected that a Java programmer understands the contents of Java bytecode. In certain situations, as in our project domain, it is helpful to understand the contents of Java bytecode in order to capture information in source code. The reason why we choose Java bytecode over plain-text source code will be discussed in Chapter 6.

We need specific tools to interpret the contents of Java bytecode and translate

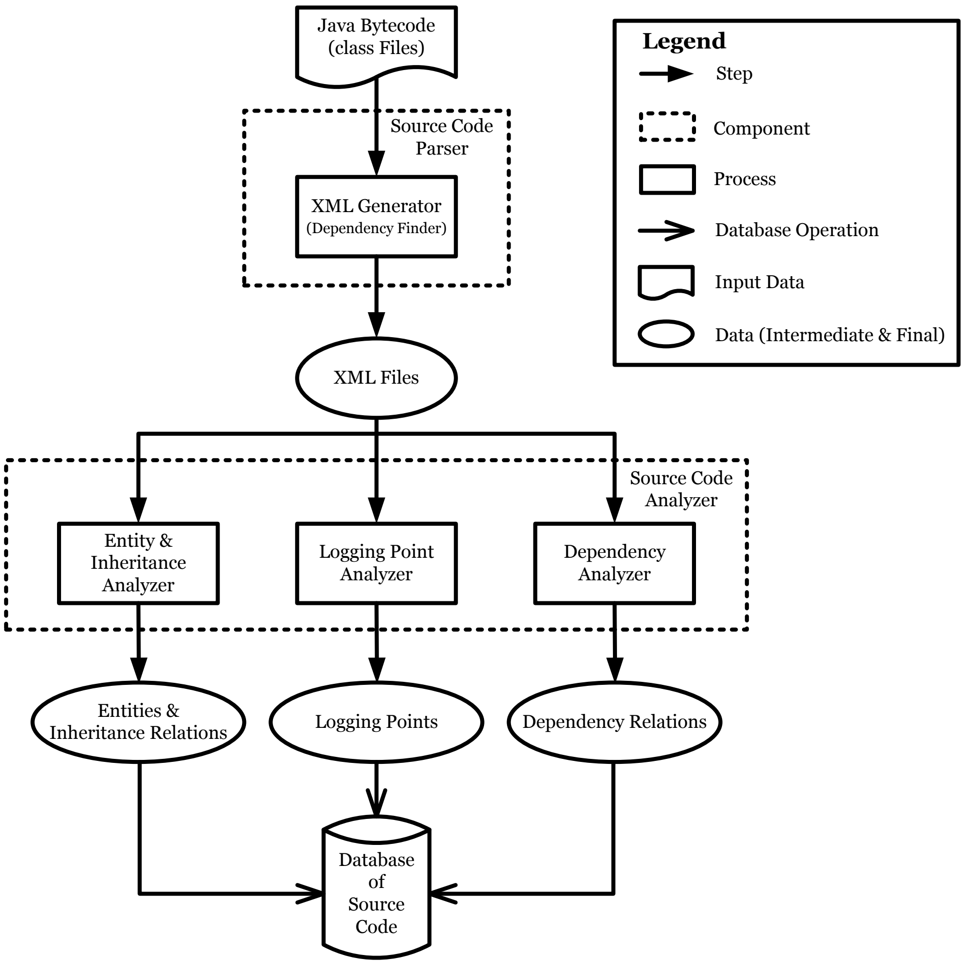


Figure 5.2: Process of Source Code Analysis

them into an XML representation. Dependency Finder is one such tool. This application generates a one-to-one XML representation of a Jave bytecode (*class*) file, which can be parsed and interpreted automatically by an XML reader program.

In summary, the source code parser takes Java bytecode as input, translates these *class* files into XML files using Dependency Finder, parses the XML for classes and methods, collects information about these entities, saves them in the form of intermediate data so that they are ready for source analysis.

Source Code Analyzer

There are several kinds of information that the source code analyzer needs to collect from Java bytecode. First, it recognizes entities, i.e., classes and methods, in both internal and external *class* files. Second, it resolves inheritance issues, if they exist, among these entities. Third, it analyzes access dependency relations among these entities, which substantially builds up the dependency graph. Finally, it identifies methods that are responsible for generating logs during program execution.

The source code analyzer needs several iterations to collect all the necessary information. As the source code parser has already translated Java bytecode into XML representations, for each iteration, the source code analyzer instructs the parser to search for required entities information in each and every XML file. Let's explain the whole process by sequential iterations as follows:

First Iteration

It identifies entities in both internal and external *class* files. It also collects information related to each of these entities, such as the signature of a method, the class in which a method is defined, the *class* file name and so on. We need to know all classes and methods that are defined and possibly involved in program executions. This builds up the fundamental knowledge of the program, which is essential later for revealing the system structure. All recognized entities are saved as intermediate data.

During this iteration, it also resolves inheritance relations among known entities, i.e., interface and superclass. Such relations are clearly stated in Java bytecode; therefore, with the XML representation, it is not a difficult task to identify the interface or superclass, if it exists, of each class being parsed. The identified inheritance relations are saved as intermediate data as well.

Second Iteration

This is the most important and challenging iteration in the source code analysis process. It performs the access dependency analysis on entities, in order to analyze dependency relations between them and eventually build up the access dependency graph to represent such relations.

For each method being parsed in the XML representation, the analyzer analyzes methods that are called by the parsed method, matches these methods with their entity information collected from the previous iteration, connects each of these methods (callee method) with the parsed method (caller method) to form dependency relations. Besides the normal caller-callee situation, the analyzer also considers the dynamic binding situation among entities, which tends to form dependency relations as well.

Once the source code analyzer finishes the dependency analysis with all XML representations, all collected dependency relations, along with the intermediate data saved from previous iterations, are stored in the database to form the knowledge base of the software system.

Final Iteration

It identifies methods that could possibly generate logs during program execution. This information is helpful in the log analysis process of matching log messages with corresponding methods in source code. In the search for such methods, it needs to be advised about the logging technique implemented in the software system, before searching for the specific methods.

For each method being parsed in the XML representation, again, the analyzer analyzes methods that are called by the parsed method. Among these callee methods, the analyzer recognizes the specific methods for logging that follow the advised format. Eventually, all recognized methods for logging are stored in the database to expand the knowledge base of the software system.

Conclusion

The source code analysis concentrates on utilizing the source code in the form of Java bytecode to extract information about entities, such as resolving inheritance relations, analyzing dependency relations and finding methods that potentially generate logs during program execution.

The entities in our project scope include classes and methods. The methods that potentially generate logs during program execution are part of the entities. The dependency relations cover all possible caller-callee relations in the program and the logging methods are always the callee methods in those corresponding relations. Since the dependency relations are the major components that build up the dependency graph later, during the log analysis process, the source code analysis process potentially links logging methods to the dependency graph. Log messages will be parsed and matched with corresponding logging methods during the log analysis process, which further links log messages to the dependency graph through corresponding logging methods.

5.2.2 Log Analysis

The goal of log analysis is to detect potential system problems reflected in logs, diagnose a sequence of logs indicating a system error at runtime, determine the program call path leading to the error log and provide an analysis result of the root cause. The analysis is carried out in multiple separate processes, which are accomplished by the log analyzer. The process of log analysis is as demonstrated in **Figure 5.3**.

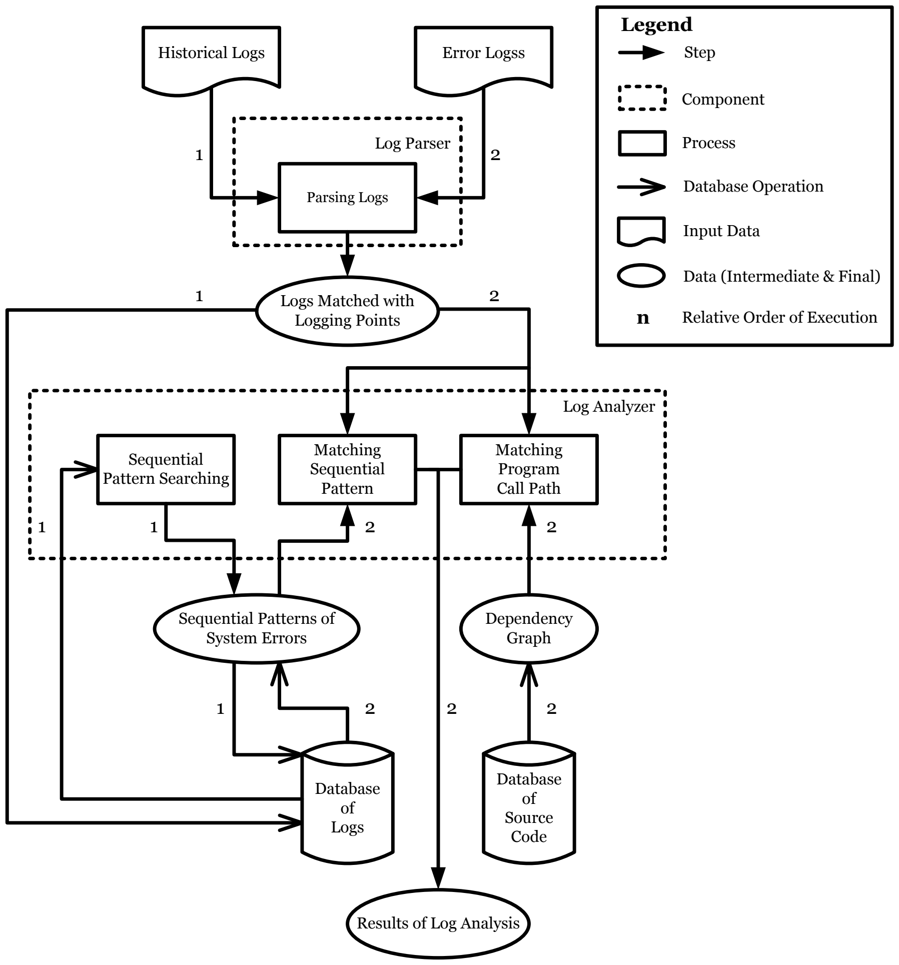

Historical Logs
Error Logss
Log Parser
Parsing Logs
Logs Matched with Logging Points
Log Analyzer
Sequential Pattern Searching
Matching Sequential Pattern
Matching Program Call Path
Sequential Patterns of System Errors
Dependency Graph
Database of Logs
Database of Source Code
Results of Log Analysis
Legend
Step
Component
Process
Database Operation
Input Data
Data (Intermediate & Final)
n Relative Order of Execution

Figure 5.3: Process of Log Analysis

Log Parser

The log parser takes a log file as input, parses every single line in the log file, to extract the timestamp of the log, the status (code) of execution, the name of the source code file being executed, the contents of the log message and so on.

It is also necessary to emphasize that most application log formats do not follow a particular standard format. This is due to the fact that logs are generated by output methods that developers insert into source code; as a result, the syntax and semantics of log contents are application-specific or even developer-specific. The log parser needs to be advised about the log format before parsing logs.

The log parser also attempts to match each parsed log with the statement or method in source code that generates it. This is an important step towards connecting the source code analysis with the log analysis, so that the log analyzer can apply both analysis techniques in the diagnosis of system problems reported by logs. The matching process makes use of information extracted from a log, and searches among eligible methods for the one that likely supplies similar information when generating the log.

Log Analyzer

The log analyzer is the core of the log analysis project. It utilizes information collected by the source code analyzer and the log parser, to analyze a given sequence of logs in two different approaches as follows:

Matching Logs with Program Call Paths

The log analyzer builds an access dependency graph to represent the dependency relations between entities, which consists of all potential program call paths. It attempts to match the given sequence of logs with the access dependency graph, in order to look for the corresponding program call path.

A program call path represents the execution sequence of methods (procedures) in source code that results in the sequence of logs; therefore, if the given sequence of logs actually indicates a system error, finding the exact program call path helps find out the method(s) along the path that may have caused the error. For each given log, the log analyzer locates the statement or method that has actually generated the log during program execution and identifies the method in the access dependency graph that calls the logging statement or method. Connecting all such matched methods gives us to the expected program call path.

Matching Logs with Sequential Patterns

The log analyzer searches for sequences of logs among historical logs, which represent system errors that have occurred in the past. We consider such sequences as sequential patterns. A sequential pattern of logs contains information about not only the execution sequence of methods in source code, but also the runtime events recorded by the logs. The process of sequential pattern searching is to study interrelations between events recorded in each pattern, collect diagnostic information about the system errors, so that the pattern can be used as reference when analyzing a similar system error in the future.

The log analyzer attempts to match the given sequence of logs with existing sequential patterns. It compares the logs of the sequence with the logs included in each sequential pattern and finds the pattern that is the closest match. If the given sequence of logs actually indicates a system error, finding the matching existing sequential pattern helps directly reveal the root cause of the error.

Conclusion

The log analysis concentrates on extracting information from historical logs and utilizing them along with information from the source code analysis to analyze a given sequence of logs. The analysis can be carried out in two different ways: matching logs with program call paths and matching logs with sequential patterns. When the given sequence of logs indicates a system error, either of these approaches can generate analysis results that are helpful to support engineers in finding the root cause.

5.2.3 Sequential Pattern Mining

Sequential pattern mining is a different process from the sequential pattern searching previously described. Sequential pattern searching mainly focuses on finding the log sequences that involve error logs and represent system errors in history, while sequential pattern mining explores potential patterns hidden in log analysis results by applying a specific mining algorithm, in order to provide an extended insight into the software system by studying interrelations among historical system errors. The process of sequential pattern mining is as demonstrated in **Figure 5.4**.

The mining process can be accomplished by running a sequential pattern mining algorithm on a set of log sequences. Sequential pattern mining is considered as a complementary approach to log analysis. It provides concrete understanding of the software system from a statistical point of view and assists support engineers in diagnosing system errors at present and in the future.

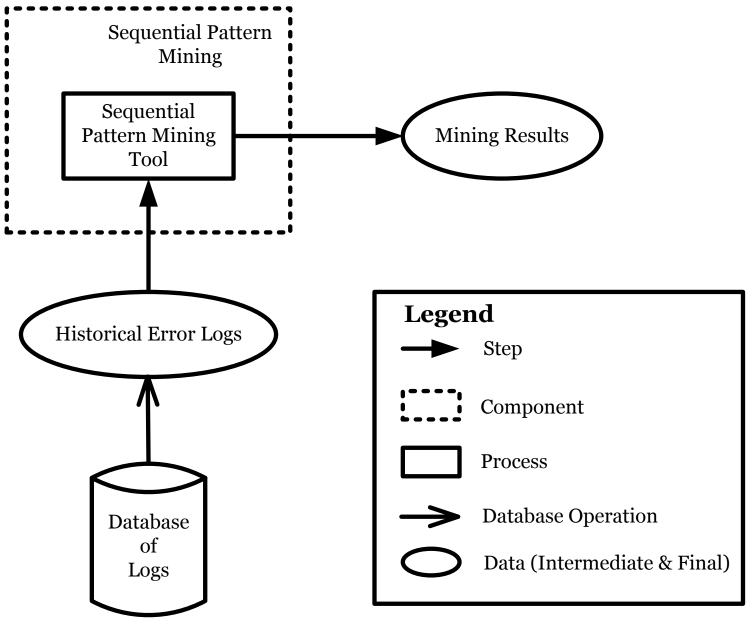


Figure 5.4: Process of Sequential Pattern Mining

5.2.4 Integration of Major Components

The source code analysis utilizes Java bytecode to recognize entities in a program and analyze access dependency relations among these entities. It also identifies methods that are responsible for generating logs during program execution. The log analysis parses log files to extract runtime information recorded among log entries. The process of matching logs with logging methods connects the source code analysis with the log analysis, so that logs in log files are linked to the corresponding logging methods in the program.

The log analysis provides two approaches to analyzing a sequence of logs indicating a system error: matching logs with program call paths and matching logs with sequential patterns. The first approach utilizes the connection between logs in log files and logging methods in the program, so that a sequence of logs can be linked to the dependency graph that is built upon the dependency relations recognized in the source code analysis. The dependency graph consists of all possible program call paths in the program. The first approach results in the program call path that potentially generates the sequence of logs being analyzed.

The second approach utilizes the sequences of historical logs related to system errors in the past. The log analysis searches for such log sequences among historical logs in the database. They are recognized as sequential patterns. Since each sequential pattern consists of a certain number of sequential logs, it can be linked to the dependency graph in the search for the corresponding program call path. The second approach matches the sequence of logs being analyzed to existing sequential patterns, which results in the expected program call path.

Sequential pattern mining is a complementary analysis that is different from sequential pattern searching in log analysis. Sequential pattern searching in log analysis mainly looks for the sequences of historical logs related to system errors in the past using database SQL query techniques. Sequential pattern mining takes all the log sequences related to past system errors as input and applies mining algorithms to explore potential sequential patterns among these log sequences. The sequential pattern in this context could be one section of the log sequence that appears more often than other sections. It implies the corresponding section of program call path and reflects one or multiple program executions that lead to the eventual system error. The mining algorithm calculates the frequencies of all recognized sequential patterns. From the statistical point of view, the frequency of a sequential pattern in the past implies the recurrence probability of the same sequential pattern; therefore the sequential patterns from log sequences related to past system errors help us better interpret any similar system errors that may occur in the future.

The integration of all major components is accomplished by another separate component called the Module Controller. The module controller handles input and outputs of all major components and coordinates tasks performed by each of them.

The module controller is demonstrated in **Figure 5.1**. It locates between the presentation tier and the rest of the components in the application tier, and it allows communication between these two tiers. It controls the application's functionality by processing incoming requests from users through the presentation tier and returning analysis results to the presentation tier. It also manages data movements among different components in the application tier.

The presentation tier accepts a user's request and passes it to the application tier, where it is received by the module controller. The module controller processes the request, decides which component in the application tier is suitable for the task, translates the incoming request to local inputs and forwards them to the suitable component. It waits for the component to finish the task, collects results as outputs from the component and, possibly, merges results from different components. Eventually, it converts the collected analysis results into the proper format and returns them to the presentation tier.

5.3 Using a Database as a Data Repository

The data tier of a three-tier architecture, as its name suggests, normally consists of a data repository where information is stored and retrieved from. The data repository can be a database or a file system, or both. For the log analysis project, we use a database as a data repository to store archive logs and analysis results, while runtime logs are stored in the form of plain-text files in the end user's file system.

There are different kinds of analysis results from either source code analysis or log analysis. We need different tables to store them separately. For example, we need to store entities and relevant information about entities in separate database tables. We also store the inheritance information into three database tables and we need two database tables for dependency information. Furthermore, we need separate database tables for data like logging methods, log messages, sequential patterns and so on.

In order to make it convenient and efficient to store and retrieve data from different database tables, we have two additional modules in the application tier to specifically handle such requests. One module is responsible for making a connection to the database and closing the connection after interactions. The other module includes all necessary methods to store, modify and retrieve data from the database.

5.4 Summary

In this chapter, we have presented an overview of the log analysis project and also discussed the principles of the three-tier architecture that we have followed in implementing all necessary components. We have also described the general process of each component and the overall integration of major components in the project.

In the next chapter, we will present implementation details of one major component of the log analysis project, i.e., source code analysis, which is one of the main subject matters of this thesis.

Chapter 6

Source Code Analysis

In this chapter, we present implementation details of the first of three major components, i.e., source code analysis, as shown in **Figure 6.1**. We start our discussion with Java bytecode. We explain how to convert Java bytecode into XML representation and how to utilize it to explore different kinds of information related to entities in the program. Then we describe the access dependency graph generation process in detail. We also discuss the algorithm and techniques that are applied in the process.

6.1 Java Bytecode

We compare Java bytecode with normal Java source code, in order to explain why Java bytecode serves more conveniently than Java source code in our project domain. We describe some Java bytecode formats that are relevant to our project. We explain why it is necessary to convert Java bytecode into XML representation, and how it could be done with the help of a third-party application.

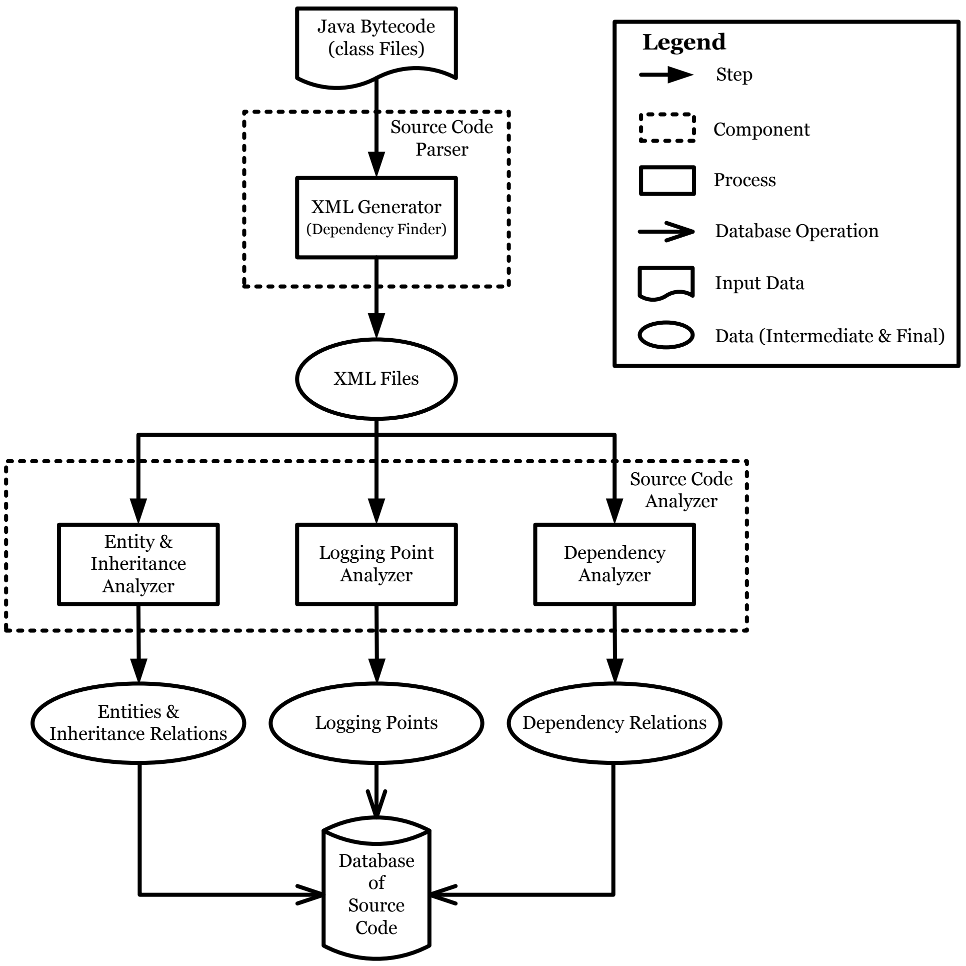


Figure 6.1: Process of Source Code Analysis

6.1.1 Java Bytecode vs. Java Source Code

The reader might wonder why we choose Java bytecode over normal Java code as the original source of our analysis. We conduct a comparison between these two representations of Java programs to demonstrate some advantages of using Java bytecode in our project domain.

- Most Java source code is written manually by programmers. Different programmers may have different coding styles, such as the comments among code lines, the indention starting code lines, the wrapping of code lines, the placement of parentheses and so on. As a result, for one Java bytecode, there is likely to be more than one presentation of Java source code. This makes it difficult to correctly parse Java source code and extract required information using string processing techniques, such as regular expressions. There is a great chance that the string processing program misinterprets the source code contents and misses out important information.

- We have discussed the analysis of dependency relations among classes and methods in a Java program. We mentioned that the inheritance relations among classes and the dynamic binding situations are big issues during the analysis process. We need to identify such relations in the program. Since Java bytecode is the compiled version of Java source code, it clearly states certain interrelations among classes, such as the superclass of a class of interface. These are helpful for our analysis. Unfortunately, Java source code does not specifically indicate such relations.

- Java bytecode can be converted into an XML representation with the help of a bytecode analysis tool. The XML representation formalizes the content of Java bytebode so that it can be conveniently parsed and interpreted with an XML reader program. Most required information, such as entities in source code and dependency relations among entities, can be accurately extracted from Java bytecode for the purpose of analysis in our project domain.

6.1.2 Format of Java Bytecode

The Java bytecode format has been fully described in The Java^{TM} Virtual Machine Specification by Sun Microsystems [LY12]. Iqbal [Iqb11] has also explained certain commonly used formats in his thesis. Each *class* file contains the definition of a single class or interface. A *class* file consists of a single `ClassFile` structure. We select an example of `ClassFile` structure from the JVM Specification [LY12]:

```
ClassFile {
   u4 magic;
   u2 minor_version;
   u2 major_version;
   u2 constant_pool_count;
   cp_info constant_pool[constant_pool_count-1];
   u2 access_flags;
   u2 this_class;
   u2 super_class;
   u2 interfaces_count;
   u2 interfaces[interfaces_count];
   u2 fields_count;
   field_info fields[fields_count];
   u2 methods_count;
   method_info methods[methods_count];
   u2 attributes_count;
   attribute_info attributes[attributes_count];
}
```

The specific items of this `ClassFile` structure that are relevant to our project [LY12, Iqb11] are:

- `access_flags` contains all access related properties of this class or interface, such as whether it is public or private, whether it is a class or interface, whether it is abstract or not and so on.
- `this_class` denotes the name of the class or interface.
- `super_class` denotes the name of the super (parent) class of the class or interface in consideration. It is worth mentioning that, if a class/interface has no super class explicitly defined in the source code file, then the default super class is `java.lang.Object`. It is also necessary to mention that Java only allows single-inheritance, so there cannot be more than one super class.
- `interfaces` denotes the set of interfaces that the class implements.
- `methods` denotes the set of methods in the class.
- `method_info` contains detailed information of all methods declared by this class or interface, including instance methods, class (static) methods, instance initialization methods, and any class or interface initialization method.

Each method, including each instance initialization method and the class or interface initialization method, is described by a `method_info` structure. No two methods in one class file may have the same name and descriptor. We select an example of `method_info` structure from the JVM Specification [LY12]:

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The specific items of this `method_info` structure that are relevant to our project [LY12, Iqb11] are:

- `access_flags` contains all access related properties of this class or interface, such as whether it is public private or protected, whether it is static or non-static, whether it is abstract or not and so on.
- `attributes` is the data structure that contains other useful information about methods. Regarding our project, we are interested in the code attribute and the signature attribute, which represent the code (instructions) and the name and parameter(s), respectively, of a method.

Java bytecode is in binary format in general. There are available bytecode analysis tools that can convert it into an XML representation, so that we are able to interpret and utilize the program information encapsulated inside Java bytecode. The one that we have used in our project is the *ClassReader* tool of *Dependency Finder* [Tes12a] application. We will provide a sample of XML representation generated by this tool.

6.1.3 The XML Representation of Java Bytecode

The *ClassReader* tool generates a one-to-one XML representation of a Java bytecode (*class*) file. We can conveniently extract all necessary information introduced in the previous subsection by parsing the XML file. Consider the class `Example`:

```
public class Example {
   public int i = 0;
   private String s = "1";
   public void a() {
      int j = b(i);
      System.out.println(s + j);
   }
   private int b(int i) {
      return i++;
   }
}
```

The XML file generated by *ClassReader* for this class is shown in **Listing 6.1**, with some of the details hidden by ellipses.

```
<classfiles>
    <classfile magic-number="-889275714" minor-version="0" major-
        version="50" access-flag="00000000 00100001">
        <constant-pool>
            ...
        </constant-pool>
        <public/>
    <super/>
    <this-class>Testing.Example</this-class>
    <superclass>java.lang.Object</superclass>
    <fields>
        <field-info access-flag="00000000 00000001">
            <public/>
            <name>i</name>
            <type>int</type>
        </field-info>
        <field-info access-flag="00000000 00000010">
            <private/>
            <name>s</name>
            <type>java.lang.String</type>
        </field-info>
    </fields>
    <methods>
        <method-info access-flag="00000000 00000001">
            <public/>
            <name>&lt;init&gt;</name>
            <signature>Example()</signature>
            <attributes>
                <code-attribute>
                    <length>16</length>
                    <instructions>
                        ...
                    </instructions>
                    <attributes>
                        ...
                    </attributes>
                </code-attribute>
            </attributes>
        </method-info>
        <method-info access-flag="00000000 00000001">
            <public/>
            <name>a</name>
            <return-type>void</return-type>
            <signature>a()</signature>
            <attributes>
                <code-attribute>
                    <length>37</length>
                    <instructions>
                        ...
                    </instructions>
                    <attributes>
                        ...
                    </attributes>
                </code-attribute>
            </attributes>
        </method-info>
        <method-info access-flag="00000000 00000010">
            <private/>
            <name>b</name>
```

```
                <return-type>int</return-type>
                <signature>b(int)</signature>
                <attributes>
                    <code-attribute>
                        <length>5</length>
                        <instructions>
                            ...
                        </instructions>
                        <attributes>
                            ...
                        </attributes>
                    </code-attribute>
                </attributes>
            </method-info>
        </methods>
        <attributes>
            <source-file-attribute>Example.java</source-file-
                attribute>
        </attributes>
    </classfile>
</classfiles>
```

Listing 6.1: XML File for `Example` Class

Even though the XML format is lengthy and complex, it is well formatted. We already stated that this is a one-to-one representation of the Java bytecode, which means all information in the bytecode, such as details of class and methods, are fully converted and presented in the XML. This XML representation can be interpreted manually, or, more importantly, can be parsed automatically by a standard XML parser.

6.1.4 Converting Java Bytecode into XML

We use the *ClassReader* tool of *Dependency Finder* application to convert Java bytecode (*class* file) into an equivalent XML representation. We need to deploy the *ClassReader* tool to our project first. The *class* file of the *ClassReader* tool is included in the *jar* file of the *Dependency Finder* application, which can be downloaded from the official website of *Dependency Finder* [Tes12a]. Once the *class* file of the *ClassReader* tool is added to our project environment, the tool is ready to use.

Common Data

We maintain a class called `GeneralData` throughout our analysis process. This holds common data and constants shared by all other classes. The common data relevant to the implementation includes:

- `classPath` - an array of Strings representing the file directories of *class* files
- `unjarOutputPrefix` - a string representing the file directory where *jar* and *zip* files are extracted to *class* files
- `classOutputPrefix` - a string representing the file directory where XML files generated from *class* files are kept

Generating XML Files

We developed a class `ClassFileParser` whose method `passClassPath` iterates through all the directories in the `classPath`. For each directory, it does two things:

1. If it finds any *jar* or *zip* file, it extracts *class* files into a subdirectory of directory `unjarOutputPrefix`. The subdirectory is named according to the *jar* or *zip* file. This is accomplished by two other methods in the `XmlGenerator` class, `lookForJarsAndZips` and `unjar`.

2. If it finds any *class* file, it generates an XML file using the `XmlGenerator` class, whose method `generateClassXml` performs the conversion by internally using the *ClassReader*. All generated XML files are placed into a subdirectory of directory `classOutputPrefix`. The subdirectory is named according to the directory where the original *class* files are located.

6.2 Finding Entities and Inheritance Relations

We continue by explaining how the converted Java bytecodes can be used to explore various kinds of information, such as entities in the program and inheritance relations among entities.

6.2.1 Definition of an Entity

In our project domain, an entity represents either a class or a method. Technically, an entity's name consists of the class name followed by a colon (:) followed by a method signature, when the entity is a method. An entity's name can be only the class name without a colon, when the entity is a class.

6.2.2 The Implementation

It is necessary to identify all existing entities and resolve inheritance relations among entities before we start analyzing the dependency relations and building the access dependency graph, because all the nodes and edges of the dependency graph are based on entities from the list of entities and the caller-callee relations between them.

Common Data

We have already introduced the `GeneralData` class, which is maintained throughout our analysis process and which holds common data and constants shared by all other classes. The common data relevant to the implementation include:

- `entityCount` - an integer representing the total number of entities
- `entityInfo` - a hash map for storing relevant information about entities
- `entity` - a hash map for storing entities that are originally defined in the program
- `inheritanceInfo` - a hash map for storing inheritance information

General Process

We developed a class `EntityParser` that identifies classes and methods in XML files and builds up the list of entities. The `parseEntity` method in this class parses each of the generated XML files and fetches out class name, superclass name, interfaces and methods. Before parsing the methods, it parses the class information and invokes the `InheritanceResolver` class to collect the superclass and interface information.

We defined a class called `ClassNode` to store information on each class, which has the following members:

```
public class ClassNode {
   String className;
   ClassNode superclass;
   HashSet<ClassNode> interfaceList;
   HashSet<ClassNode> subclassList;
   HashSet<String> transitiveChildren;
}
```

- `className` - a string representing the name of the class
- `superclass` - a reference to another `ClassNode` representing the superclass of the class
- `interfaceList` - a list of `ClassNode` representing interfaces that the class implements
- `subclassList` - a list of `ClassNode` representing the subclasses and implementations of the class
- `transitiveChildren` - a list of strings representing transitive subclasses and implementations of the class

In order to keep track of the inheritance information, we maintain a hash map data structure `inheritanceInfo` which maps an entity name to an instance of class `ClassNode` representing the entity itself.

6.2.2.1 Parsing Classes

The `parseEntity` method starts the process with parsing the class information, which is performed by the `parseClass` method. The method extracts the class name from XML and adds it to the entity list. Before calling the method `parseClassForMethod` to parse methods of the class in consideration, it invokes the `InheritanceResolver` class to collect the superclass and interface information.

6.2.2.2 Resolving Inheritance among Classes

There are four methods in the `InheritanceResolver` class. The method `checkInheritance` checks for superclass and interfaces and invokes the method `recordInheritance`, if the superclass is neither empty nor `java.lang.Object`. We do not consider `java.lang.Object` in our inheritance hierarchy because every class in Java is implicitly a subclass of it [Iqb11].

The `recordInheritance` method builds up the inheritance relation between the class in consideration and the superclass of the class. It creates instances of `ClassNode` for the class and the superclass, adds both instances to the hash map of `inheritanceInfo` and connects these two nodes based on their hierarchy. It is worth mentioning that the connection is *virtual* rather than physical. Even though instances of `ClassNode` representing each entity are placed in the hash map of `inheritanceInfo`, they are not physically linked. Instead, the inheritance connection among these entities are recorded by the data fields in each `ClassNode` instance, i.e., `superclass`, `interfaceList` and `transitiveChildren`. When accessing the `ClassNode` of any entity, it is convenient to trace corresponding hierarchy information.

The other two methods, `calculateTransitiveChildren` and `getTransitiveChildren` are used to calculate transitive children by recursively traversing the inheritance hierarchy. The information will be used for handling the dynamic binding issue during the dependency analysis process.

6.2.2.3 Parsing Methods

Returning to the `EntityParser` class after resolving inheritance among classes, we have the method `parseClassForMethod` that parses methods from XML. Like the class we parse, every parsed method is considered as an entity and added to the entity list.

6.2.2.4 Storing Entities Information in Memory

The list of entities, maintained in `entity`, is a hash map using an entity name as key and an integer as value. The entity name is in the form of either *ClassName*:*MethodSignature* for a method or *ClassName* for a class. We maintain a common data called `entityCount` that stores an integer value. The value starts from 0 and we increment it by 1 whenever we identify a new entity, either a class or a method. Then we add `entityCount` to the hash map of `entity` as the value and map it with the key in the new entity name. For example, for the following code snippet, we have the `entity` list like **Table 6.1**.

```
class A {
  public int one() {
    return 1;
  }
}
```

```
class B {

   public int two(int i) {

      return i;

   }

}
```

Entity	EID
A	1
A:A()	2
A:one()	3
B	4
B:B()	5
B:two(int)	6

Table 6.1: Sample of Entity List

It is worth mentioning that we chose hash map over other kinds of data structures because the mapping between entity name and entity id makes data query and retrieval very fast and convenient. With a specific key, we can directly access the corresponding value in the hash map. If we use a list (such as array list) instead, then every data query or retrieval requires traversal down the list, which can be time consuming when there are a large number of entities.

In addition, we maintain another hash map called `entityInfo` that stores information about entities, such as whether an entity is static or non-static. This hash map has an entity id as key and a flag as value. This flag can have the following possible values predefined in the common data class `GeneralData`:

- `CLASS (1)` if the entity is a class
- `NON_STATIC_METHOD (2)` if the entity is a non-static method

- `STATIC_METHOD (3)` if the entity is a static method
- `UNKNOWN_ACCESS_CLASS (4)` if the entity is a class but the access information is unknown, i.e., external class
- `UNKNOWN_ACCESS_METHOD (5)` if the entity is a method but the access information is unknown, i.e., external method

Although the information in this hash map is not useful for dependency graph generation, it at least provides a general understanding of classes and methods in the program.

6.3 Access Dependency Analysis

The access dependency analysis results in the access dependency graph, which will be extensively used later in log analysis. Iqbal [Iqb11] has suggested certain criteria that are taken into account during the access dependency analysis. For our log analysis project, we adopt and revise Iqbal's suggestions as follows:

1. For any two classes A and B (where A and B could possibly be the same class, or B may be an interface), if A's method $a()$ calls B's method $b()$ using any of *invokevirtual*, *invokeinterface*, *invokespecial* and *invokestatic*, then we add the following caller-callee relation to the dependency graph:

   $A : a() \rightarrow B : b()$

2. For any two classes A and B (where A and B could possibly be the same class, or B may be an interface), if A's method $a()$ calls B's method $b()$ using either

invokevirtual or *invokeinterface*, and B has transitive subclasses or implementations $B_1, B_2, ..., B_n$ explicitly implementing or overriding $b()$ as its own version, then we add the following caller-callee relations to the dependency graph in addition to the caller-callee relation described in criteria 1:

$B : b() \rightarrow B_1 : b()$

$B : b() \rightarrow B_2 : b()$

$\vdots$

$B : b() \rightarrow B_n : b()$

In addition, if B is a class that inherits method $b()$ from some other class but does not override $b()$ itself, then we add the following caller-callee relation to the dependency graph:

$B : b() \rightarrow S : b()$

where S is the closest transitive superclass of B up the inheritance hierarchy.

6.4 Access Dependency Relations

We discuss two important factors that affect the access dependency graph in a Java context: method invocation and dynamic binding issue. We clarify some terminologies that are used consistently in this thesis, before describing the process of building the access dependency relations. In the description we explain the process of parsing instructions of calling methods and describe the approach to resolving the dynamic binding issue, so that we can build up the dependency relations. Finally, we present the techniques that are used to store dependency relations in memory during analysis and in the database afterwards.

6.4.1 Considerations of Access Dependency Relations

6.4.1.1 Method Invocation

According to the Java^{TM} Virtual Machine Specification by Sun Microsystems, a method can be invoked by four different instructions [LY12]:

- *invokevirtual* invokes an instance method of an object, dispatching on the (virtual) type of the object. This is the normal method dispatch in the Java programming language.
- *invokeinterface* invokes a method that is implemented by an interface, searching the methods implemented by the particular runtime object to find the appropriate method.
- *invokespecial* invokes an instance method requiring special handling, whether an instance initialization method, a private method, or a superclass method.
- *invokestatic* invokes a class (static) method in a named class.

invokevirtual

invokevirtual dispatches a Java method. It is used in Java to invoke all methods except interface methods (which uses *invokeinterface*), static methods (which uses *invokestatic*), and the few special cases handled by *invokespecial.* Consider the code snippet:

```
Object x;
...
x.equals("hello");
```

The Java compiler generates bytecode:

```
ldc "hello"
invokevirtual java.lang.Object.equals(java.lang.Object)
```

The actual method run depends on the runtime type of the object *invokevirtual* is used with. So in the example above, if x is an instance of a class that overrides *Object*'s *equal* method, then the subclasses' overridden version of the equals method will be used [MD12].

invokeinterface

invokeinterface is used to invoke a method declared within a Java interface. Consider the code snippet:

```
public void test(Enumeration enum) {
    boolean x = enum.hasMoreElements();
    ...
}
```

invokeinterface will be used to call the *hasMoreElements* method, since *Enumeration* is a Java interface, and *hasMoreElements* is a method declared in that interface. The Java compiler generates bytecode:

```
...
invokeinterface java.util.Enumeration.hasMoreElements()
```

Which particular implementation of *hasMoreElements* method is used will depend on the type of *enum* object at runtime [MD12].

invokespecial

invokespecial is used in certain special cases to invoke a method. Specifically, it is used to invoke [MD12]:

1. the instance initialization method
2. a method in a superclass of this class
3. a private non-static method of this class

First, the main use of *invokespecial* is to invoke an object's instance initialization method during the construction phase for a new object. Consider the code snippet:

```
public void openFile(String filePath) {
    File file = new File(filePath);
    ...
}
```

The Java compiler generates bytecode:

```
...
invokespecial java.io.File.File(java.lang.String)
```

Second, *invokespecial* is also used by the Java language by the *super* keyword to access a superclass's version of a method. Consider the code snippet:

```
public class Example {
   // override equals
   public boolean equals(Object x) {
      // call Object's version of equals
      return super.equals(x);
   }
}
```

The Java compiler generates bytecode:

```
...
invokespecial boolean java.lang.Object.equals(java.lang.Object)
```

Finally, *invokespecial* is used to invoke a private non-static method. Remember that private methods are only visible to other methods belonging to the same class as the private method [MD12].

invokestatic

invokestatic calls a static method (also known as a class method) [MD12]. Consider the code snippet:

```
public class Controller {
   public static void runController() {
      sourceCodeSetup();
      ...
   }
   private static void sourceCodeSetup() {
      ...
```

```
    }
}
```

The Java compiler generates bytecode:

```
...
invokestatic void LogAnalysis.Controller.sourceCodeSetup()
```

Recognizing these four kinds of method invocations is important for analyzing calls to methods and eventually building up access dependency relations. Details of implementation will be discussed in the following sections.

6.4.1.2 The Dynamic Binding Issue

The dynamic binding issue (also known as dynamic dispatch) has been introduced in Chapter 3. It is the process of mapping a message to a specific sequence of code (method) at runtime. This is done to support the cases where the appropriate method cannot be determined at compile-time (i.e., statically). Dynamic dispatch is needed when multiple classes contain different implementations of the same method [Iqb11]. There are two kinds of implementations in a Java environment that can cause dynamic binding, class inheritance and interface implementation.

Class Inheritance

Suppose that class C extends class B which extends class A, and class A has a method $m()$ that is overridden by class B and class C, as demonstrated in **Figure 6.2**. Consider the corresponding code snippet:

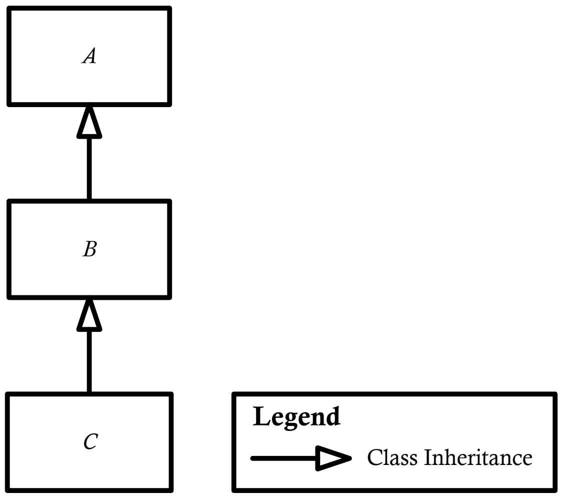


Figure 6.2: Class Inheritance

```
class A {
   public void m() {
      System.out.println("welcome");
   }
}
class B extends A {
   public void m() {
      System.out.println("hi");
   }
}
class C extends B {
   public void m() {
      System.out.println("hello");
   }
}
class D {
   public void test() {
      A a = new A();
      a.m();
      a = new B();
      a.m();
      a = new C();
      a.m();
   }
}
```

Inside the *test*() method of class D, the first call to $a.m()$ still maps to the method $m()$ in class A. The other two calls to $a.m()$ dynamically map to the method $m()$ in class B and class C, respectively, although statically both of them are bound to the method $m()$ in class A. The output of running the code snippet is as follows:

```
welcome
hi
hello
```

If class C does not override the method $m()$ while class B remains intact, and code in class D is revised:

```
class A {
   public void m() {
      System.out.println("welcome");
   }
}
class B extends A {
   public void m() {
      System.out.println("hi");
   }
}
class C extends B {
}
class D {
   public void test() {
      A a = new A();
      a.m();
      B b = new B();
      b.m();
      C c = new C();
      c.m();
   }
}
```

then the call to $c.m()$ dynamically maps to the method $m()$ in class A. The output of running the code snippet is as follows:

```
welcome
hi
welcome
```

In general, this kind of virtual method call can dynamically map to the version of that method in any other class in the inheritance hierarchy, including subclasses and superclasses [Iqb11].

Interface Implementation

Suppose that class X and class Y implement interface Z, and interface Z has a method $n()$ that is implemented by class X and class Y, as demonstrated in **Figure 6.3**. Consider the corresponding code snippet:

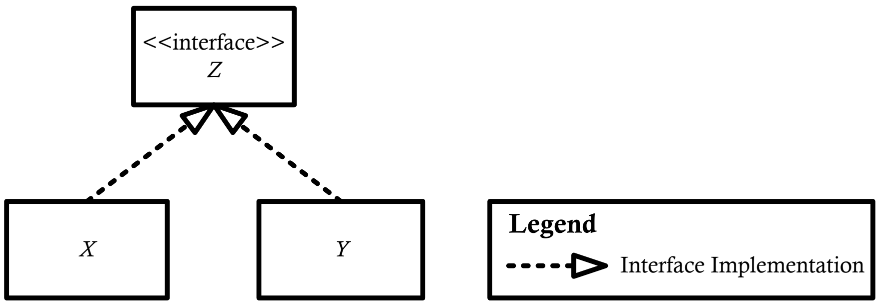


Figure 6.3: Interface Implementation

```
interface Z {
   public void n();
}
class Y implements Z {
   public void n() {
      System.out.println("Mr");
   }
}
class X implements Z {
   public void n() {
      System.out.println("Ms");
   }
}
class W {
   public void test() {
      Z z = new Y();
      z.n();
      z = new X();
      z.n();
   }
}
```

Inside the *test*() method of class W, the two calls to $z.n()$ dynamically map to the method $n()$ in class X and class Y, respectively, although statically both of them are bound to the method $n()$ in class Z. The output of running the code snippet is as follows:

```
Mr
Ms
```

Similar to class inheritance, the interface implementation can also cause dynamic binding at runtime. Since a class is allowed to implement multiple interfaces, it is more challenging to interpret the dynamic binding at runtime.

The reader might consider the above examples to be naive and unrealistic. As Iqbal [Iqb11] suggested in his thesis, the real problems in practice might be even more complicated and the potential dynamic bind issues more difficult to identify. To demonstrate that, we describe two scenarios by revisiting previous examples:

Scenario One - Class Inheritance Revisited

Going back to the example of class inheritance, we revise the code in class *D*:

```
class D {
   public void test(A a) {
      a.m();
   }
}
```

The *test*() method of class *D* receives an object of type *A* from outside as a parameter. In this case, when analyzing class *D*, there is no static way to know whether the parameter *a* represents an instance of class *A* or any subclass (direct or transitive). So it is almost impossible to interpret ahead of runtime which version of the *m*() method will be called [Iqb11].

Scenario Two - Interface Implementation Revisited

Similarly, referring to the example of interface implementation, we revise the code in class *W*:

```
class W {
   public void test() {
      Z z = new Z();
      z.n();
```

```
    }
}
```

Again, when analyzing class W, there is no way to statically interpret which version of the $n()$ method is being called.

Conclusion

We discussed two important factors that potentially affect the access dependency graph: method invocation and dynamic binding issue. We have demonstrated with examples that, the dynamic binding issue is a potential obstacle we need to overcome in order to achieve the expected log analysis result. We have also introduced four different kinds of method invocations. It is worth mentioning that, among these four instructions, only *invokevirtual* and *invokeinterface* are candidates for resolving the dynamic binding issue, while the other two are not.

6.4.2 Dependency Graph vs. Dependency Relation

The reader might find it confusing that we use *dependency graph* and *dependency relation* interchangeably in our previous discussions. Actually, both terminologies refer to the same concept, but in different contexts.

We consider the dependency relation as a caller-callee relation, or more specifically, a binary relation between two entities. For example, we have the following dependency relations:

$A : a() \rightarrow B : b()$

$A : a() \rightarrow C : c()$

We can present them in the form of binary relation, i.e., (caller, callee), then we have $\{(A : a(), B : b()), (A : a(), C : c())\}$.

On the other hand, imagine both the caller and the callee are the nodes of a graph, and the arrow ($\rightarrow$) is the edge between these two nodes in the graph, then we have a dependency graph consisting of nodes $\{A : a(), B : b(), C : c()\}$ and edges $\{(A : a(), B : b()), (A : a(), C : c())\}$.

The reader might notice that the representations of binary relations and edges of the graph are actually the same, which verifies the similarity between *dependency graph* and *dependency relation*. We will use the dependency graph to represent the dependency relations among entities during the analysis process, but when we store the dependency relations into relational database tables, they will be in the form of binary caller-callee relations.

6.4.3 Building the Access Dependency Relations

We are now ready to describe the implementation of analyzing the access dependency relation.

Common Data

We have already introduced the `GeneralData` class, which is maintained throughout our analysis process and holds common data and constants shared by all other classes. We reuse some common data introduced in the previous implementation and we need more data to hold dependency information during the process. We summarize all necessary common data for the implementation:

- `inheritanceInfo` - a hash map for storing inheritance information

- `entityCount` - an integer representing the total number of entities
- `entityInfo` - a hash map for storing relevant information about entities
- `entity` - a hash map for storing entities that are originally defined in the program
- `externalEntity` - a hash map for storing entities that are not originally defined in the program
- `nonOverriddenEntity` - a hash map for storing methods that are not overridden or implemented by other classes
- `dependency` - an array list for storing all the dependency relation
- `dynamicDependency` - an array list for storing all the dependency relations involving dynamic binding

General Process

We developed a class `DependencyAnalyzer` that iterates through all the XML files and parses the methods of a particular class for finding instructions of method invocation. To recap, a method can be invoked by four different bytecode instructions, which are *invokevirtual*, *invokeinterface*, *invokespecial* and *invokestatic*. For every instruction, we identify both caller and callee and record the relation in the common data `dependency` in terms of entity ids. Especially for instructions *invokevirtual* and *invokeinterface*, which are relevant to the dynamic binding issue, we need to analyze the potential dependency relations involving transitive subclasses and record the relations in another common data `dynamicDependency`. Eventually, all information stored in the `GeneralData` class will be saved into relational database tables.

6.4.3.1 Parsing Instructions of Method Invocation

The class `DependencyAnalyzer` has a method `buildDependencyRelation` that iterates through all the XML files and calls another method `buildDependencyRelation` to parse methods for finding dependency instructions.

For every instruction, we extract the callee class and callee method signature and compose the callee entity, then we record the caller entity and the callee entity as a dependency relation. To recap, we store the caller entity as *CallerClassName*:*CallerMethodSignature* and the callee entity as *CalleeClassName*:*CalleeMethodSignature*.

During the previous analysis process, we have stored all identified entities into an entity list using a hash map, which maps every entity name to a unique entity id; therefore, whenever we add a new dependency relation, we need to extract the corresponding ids of the caller and callee from the hash map of entity list, and place the pair of ids into the common data `dependency`, which is an array list of Java long values. The technical aspect behind this process will be explained in detail later when we discuss how to store dependency relations in memory.

Since the same method can be called multiple times by a particular method, to avoid duplication we maintain a temporary list of callee entities, called `parsedCalleeEntity`, during the parsing of each method. So if we encounter any callee entity that is already in the set, we ignore it. This actually helps us avoid unnecessary duplicate dependency relations.

In addition, it is really common to see standard Java classes and methods used in the program. For example:

```
System.out.println("Hello, world!");
```

The program calls the `println` method in the `java.io.PrintStream` class to output the string “Hello, World!” on a display console. Neither the `println` method nor the `java.io.PrintStream` class is originally defined in the program. During the analysis process, we consider such classes and methods as external entities. Separate from the `entity` list, we maintain another entity list called `externalEntity` to keep track of such entities.

6.4.3.2 Parsing Non-overridden Entities

When we discuss class inheritance and interface implementation, we mention that it is necessary to know which transitive subclasses of a class override a called method, and which transitive implementations of an interface have their own version of a called method. We need this information when we add extra dependency relations that are dynamically bound.

Iqbal [Iqb11] has suggested an intelligent approach to this problem. Instead of directly looking for methods that are overridden, he suggested looking for methods that are not overridden, then we can use the information about transitive subclasses, superclass and the non-overridden entities, to analyze the extra dependency relations relevant to the dynamic binding.

When we parse call instructions, if we find a called method that is not in our entity list but where the class to which that method belongs is in the entity list, then the class actually uses the inherited version of the method and has not overridden the method [Iqb11] . We maintain another entity list (actually another hash map) called `nonOverriddenEntity` to keep track of such entities (actually methods). We depict the following code snippet to demonstrate the idea:

```
class O {
   public void one() {
      ...
   }
   public void two() {
      ...
   }
}
class P extends O {
   public void one() {
      ...
   }
}
class Q {
   public void test(O o) {
      P p = new P();
      p.one();
      p.two();
      o.one();
   }
}
```

Suppose that class P extends class O but overrides only method $one()$, but not *two*(). So entity $P : two()$ will not be in the `entity` list. Since class Q's method calls class P's method *two*(), we add $P : two()$ to the `nonOverriddenEntity` list.

6.4.3.3 Resolving Dynamic Binding Issue

We have mentioned previously that, instructions *invokevirtual* and *invokeinterface* are related to method invocations that are dynamically bound. We need to identify them from other instructions, which leads to the following code snippet of the `DependencyAnalyzer` class:

```
...
if (   instruction.startsWith("invokevirtual") == true
     || instruction.startsWith("invokeinterface") == true) {
   isVirtualOrInterface = true;
}
...
```

Continuing with Iqbal's approach presented above, if we find a dependency relation from either an *invokevirtual* or *invokeinterface* instruction, we can use the information about transitive subclasses, superclass and the non-overridden entities, to analyze the extra dependency relations relevant to the dynamic binding. In order to separate the extra dependency relations from the ones in `dependency`, we maintain another array list of Java long values, called `dynamicDependency`.

This process is accomplished with two methods in the `DependencyAnalyzer` class. The `addToClosestSuperClass` method adds a dependency relation between the caller method and the method of the closest transitive superclass of the callee method, where the callee method is defined. This is in case the callee class has not defined the callee method. We demonstrate this process with an example. Reconsider the previous code snippet:

```
class Q {
   public void test(O o) {
      P p = new P();
      p.one();
      p.two();
      o.one();
   }
}
```

Inside class Q, method $test()$ calls method $two()$ of class P. As we have already explained, the entity $P : two()$ is a non-overridden entity, which means class P has not defined method $two()$. As a result, the call can be dynamically dispatched to method $two()$ in class O, where the method is defined. We have already added the dependency relation

$Q : test() \rightarrow P : two()$

to the list of `dependency`; in order to reflect the dynamic binding, we add the extra dependency relation

$P : two() \rightarrow O : two()$

to the list of `dynamicDependency`.

The `addImplicitAccessDependency` method adds dependency relations between the caller method and the transitive subclasses and implementations which have defined their own versions of the inherited callee method. We demonstrate this process with an example.

Consider the previous code snippet again. Inside class Q, method $test()$ calls method $one()$ of object O. However, method $one()$ has been overridden by class P. As a result, depending on the actual type of object O, the call can be dynamically bound to an instance of either class O or any transitive subclass that has also defined method one, i.e., class P. We have already added the dependency relation
$Q : test() \rightarrow O : one()$
to the list of `dependency`; in order to reflect the dynamic binding, we add the extra dependency relation
$O : one() \rightarrow P : one()$
to the list of `dynamicDependency`.

We also maintain another list `parsedDynamicCalleeEntity`, which contains the entity ids of the methods whose dynamic binding issue has been resolved, so that the methods on the list will not be considered again.

6.4.3.4 Storing Dependency Relations in Memory

We summarize what we have collected up to this point. We have an entity list and have presented each entity with a unique entity id. We have already collected a list of dependency relations and every relation consists of the caller entity id and the callee entity id. For example, for the following code snippet, we have the entity list and the dependency relations seen in **Table 6.2**.

```
class A {
   public int one() {
      B b = new B();
      return b.two();
```

```
    }
}
class B {
    public int two() {
        return 1;
    }
}
```

Entity	EID
A	1
A:A()	2
A:one()	3
B	4
B:B()	5
B:two()	6

Caller EID	Callee EID
3	5
3	6

Table 6.2: Sample of Entity List and Corresponding Dependency Relations

In the previous section, we have shown how we store the entity information in memory using a hash map. Our first approach to storing the dependency relations was using a hash map as well, which means having the caller entity id as key and the callee entity id as value. We found out very quickly that there was a big problem with this approach. Normally, the standard hash map data structure in a Java environment does not allow duplicate keys. That means, for any particular caller entity id, there can be only one matching callee entity id. If we add a dependency relation whose caller entity id already exists as a key in the hash map, the dependency relation will automatically replace the existing one. It is extremely rare that every single method in a large software system calls no more than one method. The same problem also holds when having the callee entity id as key and the caller entity id as value.

Iqbal [Iqb11] has suggested an elegant approach to solving this problem. He suggested that, in Java, an integer is 32 bits whose maximum value is more than 2000 million, which is a reasonably large amount to store all possible entities. In order to store the caller and callee entity ids as a relation in memory, we can use a Java long value which is 64 bits. So we can store the caller entity id in the upper 32 bits and the callee entity id in the lower 32 bits, as shown in **Figure 6.4**

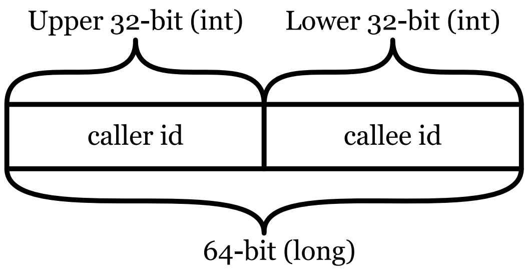


Figure 6.4: Storing Caller Id and Callee Id in Java Long Variable

So we can store the dependency relations, `dependency` and `dynamicDependency`, as two lists of Java long values. In our implementation, we have two helper methods, `encodeEntityId` and `decodeEntityId`, which encodes two entities' ids into a Java long value and decodes a Java long value back to two entities' ids. The code segment is:

```
public static long encodeEntityId(int callerEntityId, int calleeEntityId) {
   long longNum = callerEntityId;
   longNum = longNum << 32;
   longNum += calleeEntityId;
   return longNum;
}
```

```
public static int[] decodeEntityId(long longNum) {
   int[] entityId = new int[2];
   entityId[0] = (int)(longNum >> 32); // callerEntityId
   entityId[1] = (int)longNum; // calleeEntityId
   return entityId;
}
```

6.4.4 Saving Dependency Relations into the Database

Having collected all entities information and built the access dependency relations in memory, we are ready to save everything into relational database tables.

6.4.4.1 Preparing Database Tables

We need different database tables for storing different kinds of information. For storing the entities information, we have a table called `entity` with three columns: `EID`, `ClassName` and `MethodName`. `EID` stores a unique id assigned to each entity. `ClassName` and `MethodName` store the name of the class and the signature of the method, respectively, of an entity. The table is populated with data from the hash maps of `entity`, `externalEntity` and `nonOverriddenEntity`. Remember that we keep count of entities with `entityCount`. The total number of entities in `entity`, `externalEntity` and `nonOverriddenEntity` should be equal to the value of `entityCount`.

We store the meta-data about entities in a table called `entity_info`, which has three integer columns: `EID`, `EntityTypeID` and `EntityStatusID`. The `EntityTypeID` stores the value field of the hash map `entityInfo`, which tells whether an entity is static or non-static. `EntityStatusID` stores another value that tells whether an entity is external, non-overridden, or normal. This is determined by the source of an entity, which is any one of three entity lists. We keep foreign key constraints from EID, EntityTypeID and EntityStatusID to the EID column of the entity table.

For storing inheritance information, we need three different tables: inheritance for storing relations between the class and the transitive subclasses of the class, inheritance_superclass for storing relations between the class and the superclass of the class, and inheritance_interface for for storing relations between the class and the interfaces that the class implements. Each of the three tables has two integer columns: ParentClass and ChildClass, which hold the parent entity id and the child entity id, respectively. The data of these three tables are populated from the hash map of inheritanceInfo. We keep foreign key constraints from ParentClass and ChildClass to the EID column of the entity table. The pair ParentClass and ChildClass forms a primary key, so that we can guarantee each inheritance relation is unique.

Finally, we store dependency relations into two tables, dependency and dependency_dynamic, which correspond to the lists of dependency and dynamicDependency. Each table has two integer columns: CallerID and CalleeID, which hold the caller entity id and the callee entity id, respectively. We also keep foreign key constraints from CallerID and CalleeID to the EID column of the entity table. The pair CallerID and CalleeID forms a primary key, so that we can guarantee each dependency relation is unique.

6.4.4.2 Inserting Data into Database Tables

We have two classes, DatabaseConnection and DatabaseHelper to help us with the database. The DatabaseConnection class has methods that help us to make a connection to the database and to close the connection after interactions. The `DatabaseHelper` has numerous methods that specialize in adding new data (i.e., *insert*), querying interested data (i.e., *select*), modifying existing data (i.e., *update*) and removing unnecessary data (i.e., *delete*) from the database.

6.5 Finding Logging Points

We have demonstrated an approach to collecting entities (i.e., classes and methods) from Java bytecode. Some of these entities are actually involved in the process of generating logs during program execution. Identifying such entities helps us build up the connection between logs and source code in the log analysis. We start our discussion with the definition of a logging point, and we introduce one logging framework in the Java environment, Apache log4j. Finally, we describe the implementation of identifying logging points using Java bytecode.

6.5.1 Definition of a Logging Point

In our project domain, a logging point refers to a logging statement in source code that outputs a log message at runtime. The logging statement can be as simple as a *print* statement and as complicated as a statement handling a Java exception.

The simplest logging point is the *print* statement, or other equivalent output statements in source code, but in practice, it can be far more complicated than that. For example, a logging subroutine, instead of the simple logging statement, can also be used for logging. The actual logging statement, i.e., the *print* statement or other equivalent output statement, is called within the logging subroutine. When we need the program to do logging, the logging subroutine will be called; therefore the logging point should be where the logging subroutine is called in source code, rather than where the logging statement is.

An object-oriented language, like Java, applies a more complex logging mechanism in the environment, which is known as Java logging framework. In such a framework, there are predefined classes and methods for handling Java exceptions, formatting log messages and outputting the log message to a dedicated medium, such as a plain-text file. In this case, the logging point should be where the instance method of a specific logging class is called in source code. From an analysis point of view, it needs to be advised about the logging class and method that are in use, before parsing and analyzing the logging points in source code.

6.5.2 Java Logging Framework

We have already mentioned the concept of Java logging framework. We will provide more details of this concept, because that is what has been used in our partner's software system for logging. We believe the description will help the reader better understand the implementation of analyzing logging points.

A Java logging framework is a computer data logging package for the Java platform. It standardizes the process of logging in a Java environment. Normally, logging is broken into three major pieces: the Logger, the Formatter, and the Handler (Appender). The Logger is responsible for capturing the message to be logged, along with certain meta-data, and passing it to the logging framework. After receiving the message, the framework calls the Formatter with the message. The Formatter accepts the message object and formats it for output. The framework then hands the formatted message to the appropriate Handler (Appender) for disposition. This might include a console display, writing to disk, appending to a database, or email.

There are several Java logging frameworks, such as Java Logging API, Apache log4j, Apache Commons Logging and so on. The specific framework that has been used in our partner's software system is Apache log4j.

Apache log4j

Apache log4j^{TM} [Fou12c, Fou12b] is a Java-based logging package and widely used in many projects and platforms. The log4j package makes it possible to enable logging at runtime without modifying the application binary. It is designed so that these statements can remain in shipped code without incurring a heavy performance cost. Logging behaviour can be controlled by editing a configuration file, without touching the application binary [Fou12c].

Log4j has three main components: Loggers, Appenders and Layouts. These three types of components work together to enable developers to log messages according to message type and level, and to control at runtime how these messages are formatted and where they are reported [Fou12b].

Loggers are logical log file names. They are the names that are known to the Java application. Each logger is independently configurable as to what level of logging it currently logs. The actual outputs are done by Appenders. There are numerous Appenders with descriptive names available. Multiple Appenders can be attached to any Logger, so it is possible to log the same information to multiple outputs. Appenders use Layouts to format log entries. One of the most popular formats is one-line-at-a-time log file [Fou12b].

Logging Levels of Apache log4j

As previously stated, each logger can be assigned with logging levels that it currently logs. The set of possible levels includes [Fou12b]:

- The **TRACE** level designates finer-grained informational events than the DEBUG.
- The **DEBUG** level designates fine-grained informational events that are most useful to debug an application.
- The **INFO** level designates informational messages that highlight the progress of the application at coarse-grained level.
- The **WARN** level designates potentially harmful situations.
- The **ERROR** level designates error events that might still allow the application to continue running.
- The **FATAL** level designates very severe error events that will presumably lead the application to abort.

The set of logging levels are defined in the `org.apache.log4j.Level` class.

Logger Class of Apache log4j

The Logger is one of three main components of log4j. A Logger is an object that allows the application to log without regard to where the output is sent or stored. The application logs a message by passing an object or an object and an exception with an optional severity level to the logger object under a given a name or identifier.

In order to better demonstrate this idea, we present the structure of the `Logger` class and some of the basic methods [Fou12b]:

```
package org.apache.log4j;
public class Logger {
   // creation & retrieval methods:
   public static Logger getRootLogger();
   public static Logger getLogger(String name);
   // printing methods:
   public void trace(Object message);
   public void debug(Object message);
   public void info(Object message);
   public void warn(Object message);
   public void error(Object message);
   public void fatal(Object message);
}
```

Most loggers are instantiated and retrieved with the class static `Logger.getLogger` method. The reader might realize that, for each of six logging levels, there is a corresponding printing method that is named with the same wording. These printing methods are used widely for logging in source code.

6.5.3 The Implementation

After introducing all the background information about logging point and Java-based logging techniques, we are ready to describe the implementation of identifying the logging point in our partner's software system, which uses one of Java logging frameworks, Apache log4j.

6.5.3.1 Logging Point in Source Code

We have already introduced the `Logger` class and some of the logging methods of log4j. We demonstrate how they are used in source code with the following code snippet:

```
public class X {
   private static Logger log = Logger.getLogger(X.class);
   ...
   log.debug("Hello, world!");
}
```

The logger is instantiated with the `Logger.getLogger` method at the beginning of the class. Later inside the class, when logging is needed, one of those printing methods, such as `log.debug()`, is called to log the message "Hello, world!"; therefore the code line of

```
log.debug("Hello, world!");
```

is considered as a logging point in our project domain.

6.5.3.2 Parsing Logging Points in Java Bytecode

We developed a class called `SourceCodeAnalyzer` whose method `parseSourceCodeFile` iterates through all the XML files and calls another method `parseLoggingPoint` to parse methods in each class. Within each method parsed, the method `parseLoggingPoint` parses for bytecode instructions that invoke the logging methods of log4j.

Remember that we previously introduced four different instructions in Java bytecode that can invoke a method: *invokevirtual*, *invokeinterface*, *invokespecial* and *invokestatic*. We only consider instructions of *invokevirtual* and *invokeinterface* when parsing for logging points. *invokevirtual* invokes an instance method of an object, dispatching on the (virtual) type of the object. This is the case for the `Logger` class. *invokeinterface* invokes a method that is implemented by an interface, searching the methods implemented by the particular runtime object to find the appropriate method. The `Logger` class implements other interfaces in `org.apache.log4j`; therefore we includes *invokeinterface* in our consideration. The corresponding Java bytecode of the logging point in the previous example is as follows:

```
ldc Hello, world!
invokevirtual void org.apache.log4j.Logger.debug(java.lang.Object)
```

Whenever we capture either *invokevirtual* or *invokeinterface* instructions, we check whether the callee class matches with `org.apache.log4j.Logger` and the callee method name matches with one of log4j logging methods. If both match, then it confirms that the instruction represents a logging point.

6.5.3.3 Differentiating Logging Points

It is worth mentioning that the same logging method can be called more than once in a method in source code. Thus, we cannot differentiate logging points in a method by simply using the name of logging method. Originally, we considered using code line number, since every code line has a unique line number in source code. Unfortunately, Java bytecode does not contain any information about the code line number. We had to turn our focus back to the logging methods. We found that it is very common for software developers to insert a constant string into a log message, so that it is easier for support engineers to connect the actual log message in log files with the logging method in source code. Considering this observation, we decided to include the constant string in the metrics of differentiating logging points.

The reader might wonder, can two logging points in the same method have the same constant string in their log messages? The answer is Yes! But in practice, it is difficult for support engineers to differentiate them or match them with corresponding log messages. It is good practice for software developers to avoid such a programming style. In our project domain, we have to make this assumption, which means all logging points in the same method should have different constant strings in their log messages. Logging points in different methods can have the same constant string since they can be differentiated by their caller methods.

Reconsider the previous code snippet. Before the *invokevirtual* instruction, there is another *ldc* instruction, which pushes a one-word constant onto the operand stack [MD12]. The one-word constant, i.e., "Hello, world!", is popped up later and inserted into the log message by instruction *invokevirtual*. It is obvious that the one-word constant is exactly the constant string we are interested in. So inside the method `parseLoggingPoint`, we have a temporary variable to keep track of the constant pushed by the *ldc* instruction before the *invokevirtual* or *invokeinterface* instruction. When we find a logging point, the corresponding constant string will be stored along with the logging point.

In summary, the metrics used to differentiate logging points include the constant string in the log message, the logging method, the method where the logging method is called, i.e., the caller method, and the Java source code file where the caller method is defined.

6.5.3.4 Storing Logging Points

Storing in Memory

We defined a class called `LoggingPoint` to store information about a logging point, which has the following members:

```
public class LoggingPoint {
   int lid;
   int eid;
   int callerEid;
   String sourceCodeFile;
   String logContent;
}
```

- `lid` - an integer representing an unique id assigned to each logging point
- `eid` - an integer representing the entity id of the logging method
- `callerEid` - an integer representing the entity id of the method calling the logging point
- `sourceCodeFile` - a string representing the name of the source code (*java*) file
- `logContent` - a string representing the constant in the log message

During the analysis process, we maintain an array list of `LoggingPoint` instances to store all collected logging points in memory.

Remember that the logging class `Logger` and all logging methods used in source code are considered entities as well; therefore each of them has an unique entity id. The source code file is the plain-text *java* file, not the Java bytecode *class* file or the XML file. The reason behind this will become obvious in Chapter 7 when we describe the process of matching logs with logging points.

Saving into the Database

For storing information about logging points, we prepare a database table called `logging_point` with five columns: `LID`, `EID`, `CallerEID`, `SourceCodeFile` and `LogContent`. `LID` stores a unique id assigned to each logging point. `EID` stores the entity id of the logging method. `CallerEID` stores the entity id of the caller method of the logging point. `SourceCodeFile` stores the name of the source code (*java*) file. `LogContent` stores the value of the constant string used in the log message. We keep foreign key constraints from `EID` and `CallerEID` in the `EID` column of the `entity` table. The `LID` is a primary key. The table is populated with data from the array list of `LoggingPoint` instances in memory.

6.6 Summary

In this chapter, we have presented implementation details of source code analysis, which is the first of three major components in our log analysis project. We discussed concepts of Java bytecode and demonstrated how we utilize the XML representation of Java bytecode to explore necessary information about a software system, i.e., entities information, inheritance relations among entities and logging points in source code. We discussed the concepts of the access dependency graph and access dependency analysis. We described the process of building the access dependency relations. These are the building blocks of the eventual log analysis.

In the next chapter, we will present implementation details of another major component in the log analysis project, i.e., log analysis, which is another main subject matter of this thesis.

Chapter 7

Log Analysis

In this chapter, we present implementation details of the second of three major components, i.e., log analysis, as shown in **Figure 7.1**. We describe a general process of parsing logs in log files, extracting log information and matching each log with the corresponding logging point. Then we explain how to utilize the access dependency relations to build up the access dependency graph for eventual log analysis. We also describe the process of searching for sequential patterns among historical logs. The final analysis includes two different approaches. One approach is to match a sequence of logs with the access dependency graph and the other is to match the log sequence with existing sequential patterns of logs.

7.1 Parsing Logs

Logs of the software system are stored in plain-text files. As previously discussed, it is more convenient and efficient to manage data in a relational database management system than in a conventional storage medium, i.e., text file; therefore, before

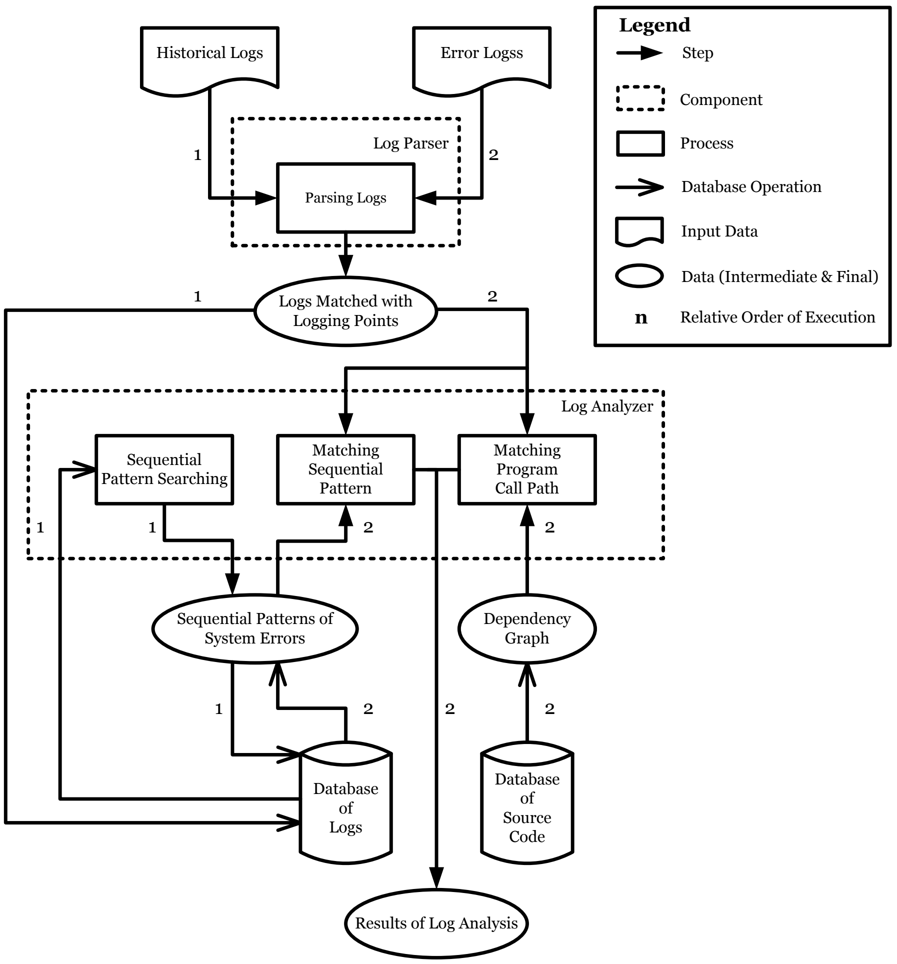

Historical Logs
Error Logss
Legend
Step
Component
Process
Database Operation
Input Data
Data (Intermediate & Final)
n Relative Order of Execution
Log Parser
Parsing Logs
Logs Matched with Logging Points
Log Analyzer
Sequential Pattern Searching
Matching Sequential Pattern
Matching Program Call Path
Sequential Patterns of System Errors
Dependency Graph
Database of Logs
Database of Source Code
Results of Log Analysis
1
2

Figure 7.1: Process of Log Analysis

performing any analysis of logs, we transfer them from the log file to the database. During the process, we parse logs, extract necessary information from log messages, and most importantly, match each log with the corresponding logging point in source code. Finally, we save all of the collected information into database tables for future reference.

7.1.1 The Log Format of Our Partner's System

Since most application log formats do not follow a particular standard format, the log format used in a software system needs to be known before parsing log files. The information advised may include how many fields there are in a log, what kind of information each field represents, what is the delimiter used to separate these fields and so on. We have mentioned that the logging framework used in our partner's software system is Apache log4j. One sample log output by Apache log4j is as follows:

```
DEBUG 2012-02-29 12:34:56,789 [main] Testing.X - Hello, world!
```

- `DEBUG` - This field represents the log level. It can be one of the six log levels in Apache log4j, which are `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR` and `FATAL`, in increasing order of severity.

- `2012-02-29 12:34:56,789` - This field represents the date and time when this log is generated.

- `[main]` - This field represents information about the thread that runs the program and generates this log.

- `Testing.X` - This field represents the Java class where the logging method is called to generate this log.

  In general, the name of a Java class is the same as the file name where the class is defined. It is safe to assume that this field also represents the name of Java source code file without the *.java* extension.

- `Hello, world!` - This field represents the contents of the log message.

It is worth mentioning that, as part of the log format, there is a space between each of these fields in a log. Even though there is also a space separating the date and the time, we still consider them as one field representing a *timestamp*. There is a dash (-) between the Java class and the contents of the log message.

7.1.2 Extracting Information from a Log

Now we understand the log format, we are ready to parse logs in the log files and extract information from the fields in each log. We developed a class called `LogParser` whose method `parseLogFile` iterates all the log files in the system and calls another method `openLogFile` to parse the logs in each log file.

The log file generated by Apache log4j adopts the one-line-at-a-time format, which means every log takes up one line without wrapping in the log file; therefore the method `openLogFile` reads the log file line by line and passes it to another method `parseLogMessage` to process the log contents. For each log, the `parseLogMessage` method applies regular expression technique to match the log with the expected log format.

It is worth mentioning that a regular expression provides a concise and flexible means for *matching* (specifying and recognizing) strings of text, such as particular characters, words, or patterns of characters. A regular expression is usually written in a formal language that can be interpreted by a regular expression processor, which is a program that either serves as a parser generator or examines text and identifies parts that match the provided specification. In Java, a class called `Pattern` works as the regular expression processor and compiles a regular expression, specified as a string, into an instance of this class. The resulting pattern can be used to create an instance of `Matcher` class, which matches arbitrary character sequences against the regular expression. The `Matcher` class works as the parser that performs match operations on the character sequences by interpreting the pattern. In our project, the regular expression used to match logs is as follows:

```
static String logMsgRegex = "^([FATAL|ERROR|WARN|INFO|DEBUG|TRACE])
\\s{1,2}(\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2},\\d{3})
\\s(\\[.+\\]|\\[{2}.+\\].+\\])\\s(\\w+[\\.*\\w*]*)\\s*-\\s(\\s*\\w.+)$";
```

and the code segment that performs regular expression matching is as follows:

```
Pattern pattn = Pattern.compile(RegexHelper.logMsgRegex);
Matcher matcher = pattn.matcher(logLine);
```

One can divide the character sequences into different groups using parentheses in a regular expression. In the previous example, we divide the regular expression of expected log format into five different groups, each of which represents one of the fields in a log.

Upon a successful match, the method `parseLogMessage` extracts information from the matched log, with the help of the `Matcher` class. The matched subsequences are captured and temporarily stored in the capturing groups. The `Matcher` class has methods for accessing these capturing groups and returning the captured subsequences in string form. The returned information is stored into an instance of `Log` class.

7.1.3 Matching a Log with a Logging Point

When a log is matched with the regular expression, the `parseLogMessage` method passes the log to another method `matchLoggingPoint`, which attempts to match the log with the corresponding logging point in source code.

We have collected information about logging points during the source code analysis. To recap, for each logging point we learnt about the constant string in the log message, the logging method, the method where the logging method is called, i.e., the caller method, and the Java source code file where the caller method is defined.

The `matchLoggingPoint` method retrieves existing logging points of the source code file where the log is generated. It iterates through the list of logging points and performs match operations on each of them. We know the corresponding logging method from the log level recorded. That helps narrow down the list of eligible logging points. Then it compares the constant string of the logging point with the message contents in the log. If the constant string appears in the log message, then the match is confirmed. If that is the case, the `matchLoggingPoint` method stores relevant information about the matched logging point into the instance of `Log` class and returns a positive confirmation; otherwise, it simply returns a negative confirmation.

7.1.4 Storing the Log Information

Storing in Memory

We have mentioned the `Log` class several times in previous descriptions. We developed this class to store information about the log plus the corresponding logging point in memory. The `Log` class has the following members:

```
public class Log {

   int mid;

   int lid;

   int eid;

   int callerEid;

   String logFile;

   int logLine;

   String status;

   String logTime;

   String thread;

   String sourceCodeFile;

   String message;

   boolean matched;

}
```

- `mid` - an integer representing a unique id assigned to each log message
- `lid`, `eid` and `callerEid` - representing relevant information about the logging point that generates this log
- `logFile` - a string representing the full file directory of the log file
- `logLine` - an integer representing the relative line number of this log in the log file
- `status`, `logTime`, `thread`, `sourceCodeFile` and `message` - representing information extracted from the fields of this log

- `matched` - a boolean representing the confirmation that the corresponding logging point is found

During the analysis process, we maintain an array list of `Log` instances to store all parsed logs in memory.

Saving into the Database

For storing logs parsed from log files, we prepare a database table called `log_message` with nine columns. `MID` stores a unique id assigned to each log. `LID` stores the id of the logging point that generates the log. `LogFile` stores the full file path of the log file. `LogLine` stores the relative line number where the log appears in the log file. `Status`, `LogTime`, `Thread`, `SourceCodeFile` and `Message` store information extracted from the log. The `MID` is a primary key. For the purpose of efficient data query, we have the table indexed with `LID`. The table is populated with data from the array list of `Log` instances in memory.

7.2 Access Dependency Graph

The implementation of connecting logs in the log file with logging points in source code puts us one step closer to the final log analysis process. That is only for one individual log. In order to analyze a sequence of logs, we still need to implement an approach to expand this individual matching to a sequential matching, i.e., matching between a sequence of logs and logging points in a specific order. We have described the process of analyzing the access dependency relations. We will explain how to utilize these relations in our analysis process. We present two graph search algorithms that help us efficiently traverse the dependency graph.

7.2.1 Dependency Graph vs. Dependency Relation - Revisited

We have clarified the similarities and differences between the two terminologies *dependency graph* and *dependency relation*. To recap, they refer to the same concept, but in different contexts.

When we describe the process of analyzing interrelations among entities in source code analysis, i.e., caller-callee relations, we use the terminology *dependency relation* more often because it properly represents the binary relation between two entities which are caller and callee, respectively. When we store the caller-callee relations into relational database tables, technically they are dependency relations, but conceptually, they are also a dependency graph.

In this chapter, we describe our analysis process in the domain of abstract graphs. We utilize the dependency relations in database tables to build up the dependency graph in memory. We present two graph search algorithms and discuss the implementation of applying them to traverse the dependency graph. It is more convenient and specific to use the terminology *dependency graph* in our descriptions.

7.2.2 Building the Access Dependency Graph

We developed a class `Graph` that connects dependency relations in database tables with edges and builds up the access dependency graph in memory. We also developed another class `GraphNode` that represents each entity as a node of the dependency graph. The class `GraphNode` has the following members:

```
public class GraphNode {
   int entityId;
   String entityName;
   ArrayList<GraphNode> callerList;
   ArrayList<GraphNode> calleeList;
}
```

- `entityId` - an integer representing the id of an entity
- `entityName` - a string representing the name of an entity, either *ClassName* for a class or *ClassName*:*MethodSignature* for a method
- `callerList` - an array list representing all the caller entities of an entity
- `calleeList` - an array list representing all the callee entities of an entity

A dependency graph built on the caller-callee relations is a directed graph. The directed edge (arrow) can either point from caller entity to callee entity or vice versa. We decide to connect the entities in a way such that the directed edges point from callee entities to caller entities.

Inside the `Graph` class, there is a method `buildDependencyGraph` that obtains all dependency relations from the database and iterates through all the relations to connect entities with directed edges. Remember that there are two kinds of dependency relations previously discussed. One is the normal caller-callee relation and the other is the access relation involving dynamic binding consideration. So the `buildDependencyGraph` method first iterates through the list of dynamic dependency relations and identifies caller and callee entities involving the dynamic binding issue, then iterates through another list of normal dependency relations, builds directed edges from callee entities to caller entities and further adds directed edges to callee entities in the dynamic binding relations. Once all entities in the dependency relations are connected with directed edges, the dependency graph is completed.

7.2.3 Traversing the Access Dependency Graph

Graph traversal refers to the problem of visiting all the nodes in a graph in a particular manner. It explores the structure of the graph and discovers the connections between the nodes. Depth-first Search (DFS) and Breadth-first Search (BFS) are two techniques for traversing a finite graph.

7.2.3.1 Breadth-first Search (BFS)

Given a graph $G = (V, E)$ and a distinguished *source* vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s [CCRS09].

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$ [CCRS09].

From the standpoint of the algorithm, all successor nodes obtained by expanding the predecessor node are added to a *First In, First Out* (FIFO) queue for exploration. The pseudo-code of a breadth-first search algorithm is shown in **Algorithm 1**.

Algorithm 1 Breadth-first Search

```
function BFS($G$, $v$)
    Queue $Q$ := {};
    for all $u$ in $G$ do
        $visited[u]$ := False;
    end for
    add $Q$, $v$;                         ▷ add to the tail of the queue
    while $Q$ is not empty do
        $u$ := remove $Q$;                ▷ remove from the head of the queue
        if not $visited[u]$ then
            $visited[u]$ := True;
            for all neighbour $w$ of $u$ do
                if not $visited[w]$ then
                    add $Q$, $w$;
                end if
            end for
        end if
    end while
end function
```

7.2.3.2 Depth-first Search (DFS)

Depth-first search explores edges out from the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex [CCRS09].

In a non-recursive implementation, all successor nodes obtained by expanding the predecessor node are added to a *Last In, First Out* (LIFO) stack for exploration. The pseudo-code of a depth-first search algorithm without recursion is shown in **Algorithm 2**.

Algorithm 2 **Depth-first Search**

```
function DFS($G$, $v$)
    Stack $S$ := {};
    for all $u$ in $G$ do
        $visited[u]$ := False;
    end for
    push $S$, $v$;                              ▷ push to the top of the stack
    while $S$ is not empty do
        $u$ := pop $S$;                         ▷ pop from the top of the stack
        if not $visited[u]$ then
            $visited[u]$ := True;
            for all $w$ adjacent to $u$ do
                if not $visited[w]$ then
                    push $S$, $w$;
                end if
            end for
        end if
    end while
end function
```

Different from BFS, DFS starts at the selected root node and explores as far as possible along each branch. It traverses deeper and deeper until either a goal node is found or it reaches a node that has no successor; therefore a recursive implementation of DFS is feasible. The pseudo-code of a depth-first search algorithm with recursion is shown in **Algorithm 3**.

7.2.3.3 BFS vs. DFS

BFS visits the sibling nodes before visiting the child nodes, that is, it traverses the breadth of the graph before the depth. BFS is a *conservative* and *prudent* search, because it progresses equally in all possible paths. Given a graph $G = (V, E)$, the total running time of the BFS procedure is $O(V + E)$ [CCRS09].

DFS visits the child nodes before visiting the sibling nodes, that is, it traverses

Algorithm 3 Depth-first Search with Recursion

```
function BFS($G$, $v$)
    for all $u$ in $G$ do
        $visited[u]$ := False;
    end for
    $BFS\_Recusion(v)$;
end function
function BFS_Recusion($v$)
    $visited[v]$ := True;
    for all $w$ adjacent to $v$ do
        if not $visited[w]$ then
            $BFS\_Recusion(w)$;
        end if
    end for
end function
```

the depth of the graph before the breadth. DFS is an *aggressive* and *risky* search, because it simply chooses one path and ignores all other paths until it reaches the end of the chosen path. It is possible for it to reach all the other nodes in the graph before reaching the goal node. Given a graph $G = (V, E)$, the total running time of the DFS procedure is $\Theta(V + E)$ [CCRS09].

DFS has the advantage over BFS in terms of memory usage as it has a much lower memory requirement than BFS. Since BFS has to store all of the successor nodes at each level, in the worse case, it needs to store $O(2^n)$ nodes at level n in the graph. On the other hand, DFS only has to store the current path from the starting node to the current node that it is expanding, so, in the worst case, it needs to store only $O(n)$ nodes at level n in the graph.

Either BFS or DFS can be efficient when searching for a particular node in the graph, depending on the structure of the graph and the location of the node. For example, if the goal node is at the lower level of the graph, then BFS usually takes longer to reach the node than DFS. If the goal node is at the upper level of the graph, then BFS is usually faster than DFS.

From the standpoint of implementation, BFS implements a *First In, First Out* (FIFO) strategy such that it appends new nodes to the tail of a queue and draws nodes from the head of the queue for exploration. DFS implements a *Last In, First Out* (LIFO) strategy such that it pushes new nodes to the top of a stack and pops nodes also from the top of the stack for exploration.

We demonstrate the differences between the two algorithms with an example. Consider a sample graph in **Figure 7.2**:

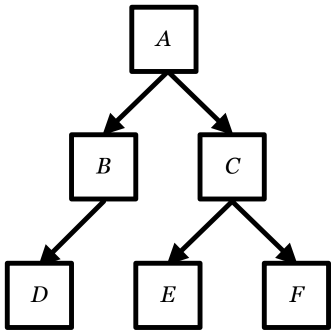


Figure 7.2: Sample Graph

BFS - The algorithm visits the nodes in the following order:

A, B, C, D, E, F

DFS - The algorithm visits the nodes in the following order:

A, C, F, E, B, D

It is assumed that new nodes are pushed onto the stack from left to right. For example, at level 2, node B is pushed onto the stack before node C; as a result, node C is popped from the stack before node B, which leads to the visiting sequence as shown.

DFS with Recursion - The algorithm visits the nodes in the following order:

A, B, D, C, E, F

7.3 Matching Logs with Program Call Paths

We are ready to describe the first of two different approaches to analyzing a sequence of logs. This approach attempts to match a sequence of logs with all of the eligible program call paths, in search of the specific one that represents the sequential program executions resulting in the sequence of logs.

7.3.1 General Process

Our ultimate goal is to analyze a sequence of logs by matching the logs with logging points in source code, so that we can reveal the sequence of program executions that leads to the occurrence of system error indicated by logs. The sequence of program executions can be represented by a program call path.

Our implementation adopts the static analysis approach. We analyze the dependency relations among entities in the program and build up the dependency graph to represent all potential program call paths, each of which is actually a subgraph of the dependency graph; therefore the general process is to determine a start node for traversing the dependency graph, match each visited entity node with the corresponding log in the given sequence and build up program call paths that consists of matched nodes. The program call paths represent the sequential program executions resulting in the given sequence of logs.

7.3.2 Determining the Start Node of the Graph Traversal

Generally speaking, when we analyze a sequence of logs that indicates a system error, our focal point intuitively falls on the log with an error status (code). We normally choose this error log as the start point of manual analysis. We also do this in the automatic approach in our project: we match the error log with the corresponding logging point in source code, identify the entity of matched logging point as a node of the dependency graph, and traverse the dependency graph starting from this node in search for the program call paths.

It is worth mentioning that, even though the logging point is identified as the start node, specifically, the actual graph traversal starts from the method that calls the logging point. We can explain this in brief. We mentioned previously that the logging point is also an entity in the program, which represents one of the six logging methods in Apache log4j. Since these logging methods are widely used in the program, if we traverse the dependency graph starting from one of them, it will include other program call paths that are not necessary for our consideration. That is the reason why we include the caller method of the logging point to differentiate between logging points. It also explains why we start the actual graph traversal from the caller method of the logging point instead of the logging point itself. This helps narrow down the range of eligible program call paths.

7.3.3 Direction of the Graph Traversal

Reconsider the example of manual analysis of a sequence of logs. After we determine the start point of analysis as the error log, we trace *backwards* the logs recorded before the error log for further investigation. The logs in a log file are organized in chronological order, which means the logs appearing before the error log represent the events (executions) that occur before the system error. The approach of tracing *backwards* in reverse chronological order usually helps us identify the suspicious logs that potentially indicate the root cause.

The same theory applies in the automatic log analysis process. We determine the logging point that generates the error log and build up program call paths from the dependency graph that starts from the node of the logging point. In order to recognize the specific program call path that corresponds to the sequence of logs, we traverse each path in a *backward* direction, i.e., from callee to caller, match the nodes at each level along the way with the log at the corresponding level, filter out the unmatched branch and look for the expected program call path.

7.3.4 Traversing with Depth-First Search (DFS)

The BFS algorithm visits the sibling nodes before visiting the child nodes, which means it traverses the breadth of the graph before the depth. The DFS algorithm visits the child nodes before visiting the sibling nodes, which means it traverses the depth of the graph before the breadth. The choice of graph traversal algorithm is determined by our approach to processing the sequence of logs.

We build up the program call paths when traversing the dependency graph. We must traverse to the end of an eligible program call path and attempt to match all of the nodes along the way with the sequence of logs, before determining whether this path matches with the log sequence or not; therefore the DFS algorithm is more suitable to our approach than the BFS algorithm. The DFS algorithm traverses to the end of a branch before traversing the next branch, it allows us to backtrack to any of the visited nodes. This is a helpful technique as we have to check back whether the path includes the required matched nodes after visiting an eligible program call path.

7.3.5 Building the Program Call Paths

Keeping Track of Traversals with Cursors

We build up the program call paths in memory by cloning and adding every visited node to the corresponding path, when traversing the dependency graph with the depth-first search algorithm. The process involves two separate traversals, one is traversing the dependency graph and the other is visiting logs in the given log sequence; therefore, accordingly, we maintain two cursors during the process. One cursor keeps track of the level, or the depth, of the program call path. It starts with level 0 and will be reset when the process finishes traversing one path. The other cursor keeps track of the currently visited log. It starts with the last log in the given log sequence and will also be reset when the process finishes traversing one path.

The process traverses the dependency graph from the selected start node, clones every visited node and adds it the corresponding program call path. It also matches every visited node with the currently visited log in the log sequence. If the node matches with the log, the log cursor moves on to the next log in the log sequence; otherwise, the log cursor remains intact until a matched node is found. The graph cursor moves on to the next node along the path no matter what the matching outcome.

Methods without Logs

The reason behind such design is that, in practice, methods could be executed without generating logs. This completely depends on the design of the methods. Some methods are not programmed to generate a log at all, which means there is no logging point in the method; some methods have logging points, but due to the restrictions of conditional statements, none of the logging points is executed and as a result, no log is generated.

Rules of Matching with Program Call Paths

We have considered both scenarios and implemented our solution to handle them accordingly. When a visited node does not match with the currently visited log, we mark it as *unmatched*. We will match the next visited node with the same log until a matched node is found, then we mark that node as *matched*. After reaching the end of the current path, we backtrack to all of the *matched* nodes. As long as the sequence of matched nodes along the program call path corresponds with the sequence of logs, no matter how many *unmatched* nodes are lying between the matched ones, we consider this program call path matches with the sequence of logs.

We remove the unmatched program call paths once we have determined whether the current program call path matches the sequence of logs by backtracking. We repeat this process with all eligible paths we encounter during graph traversal. The result will include all of the matched program call paths.

7.3.6 Examples of Matching Logs with Program Call Paths

Example One

We have a sample program with the dependency graph as demonstrated in **Figure 7.3**. The digits in each of the method names represent the corresponding level in the dependency graph. Each of all the possible call paths results in an error log. It is assumed that each method in the program generates a unique log representing the method itself when it is executed.

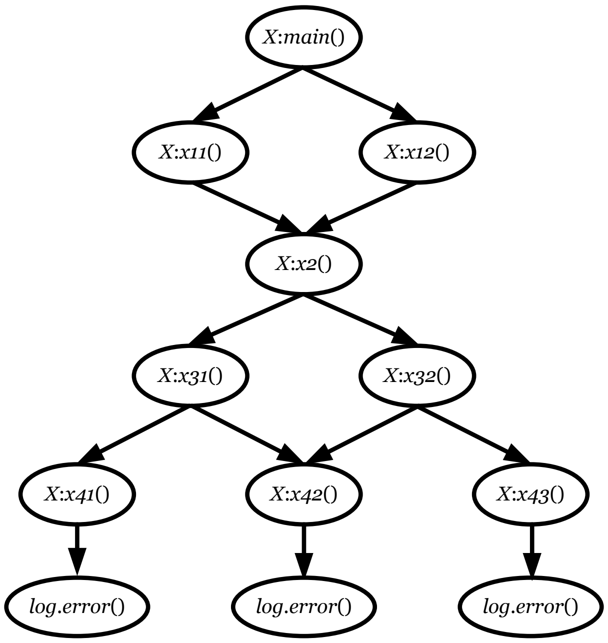


Figure 7.3: Access Dependency Graph of Example One

We have a sequence of logs involving an error log as follows:

```
DEBUG 2011-11-11 09:17:20,941 [main] Testing.X - x12
DEBUG 2011-11-11 09:17:21,956 [main] Testing.X - x2
DEBUG 2011-11-11 09:17:22,972 [main] Testing.X - x32
DEBUG 2011-11-11 09:17:23,988 [main] Testing.X - x43
ERROR 2011-11-11 09:17:24,050 [main] Testing.X - system error!
```

For the purpose of debugging, our implementation traverses the dependency graph using the Depth-first Search (DFS) algorithm and searches for all possible program call paths leading to the error log. The result is as follows:

```
Depth-First Search of Dependency Graph
Level 5 - Testing.X:main(java.lang.String[])
Level 4 - Testing.X:x11()
Level 5 - Testing.X:main(java.lang.String[])
Level 4 - Testing.X:x12()
Level 3 - Testing.X:x2()
Level 2 - Testing.X:x32()
Level 1 - Testing.X:x43()
Level 0 - org.apache.log4j.Logger:error(java.lang.Object)
```

The traversal can use either BFS or DFS. The output from these two algorithms is correct in terms of content, but slightly different in terms of appearance. The result of traversal using Breadth-first Search algorithm is as follows:

```
Breadth-First Search of Dependency Graph
Level 0 - org.apache.log4j.Logger:error(java.lang.Object)
Level 1 - Testing.X:x43()
Level 2 - Testing.X:x32()
Level 3 - Testing.X:x2()
Level 4 - Testing.X:x11()
Level 4 - Testing.X:x12()
Level 5 - Testing.X:main(java.lang.String[])
Level 5 - Testing.X:main(java.lang.String[])
```

Our implementation performs the matching process to search for the call path that has generated the sequence of logs. The result of the matching process is as follows:

```
Call Path Matching: Closest Matched Call Path
Level 4 - Testing.X:x12()
Level 3 - Testing.X:x2()
Level 2 - Testing.X:x32()
Level 1 - Testing.X:x43()
Level 0 - org.apache.log4j.Logger:error(java.lang.Object)
```

The correctness of the result can be verified with the dependency graph in **Figure 7.3**.

Example Two

We have another sample program with the dependency graph as demonstrated in **Figure 7.4**. The digits in each of the method names represent the corresponding level in the dependency graph. Each of all the possible call paths results in an error log. It is assumed that each method in the program generates a unique log representing the method itself when it is executed, except node `X:x13()` and node `X:x33()`, which do not generate a log at all during program execution.

We have the same sequence of logs involving an error log:

```
DEBUG 2011-11-11 09:17:20,941 [main] Testing.X - x12
```

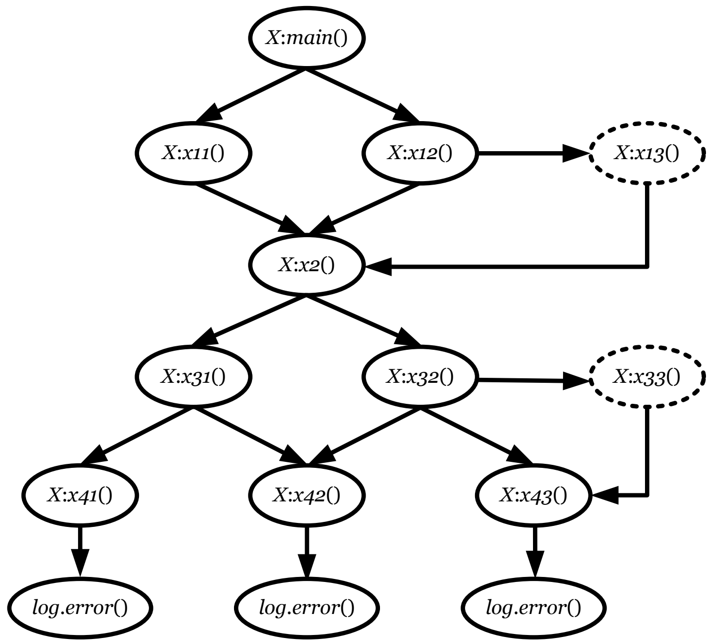


Figure 7.4: Access Dependency Graph of Example Two

```
DEBUG 2011-11-11 09:17:21,956 [main] Testing.X - x2

DEBUG 2011-11-11 09:17:22,972 [main] Testing.X - x32

DEBUG 2011-11-11 09:17:23,988 [main] Testing.X - x43

ERROR 2011-11-11 09:17:24,050 [main] Testing.X - system error!
```

Our implementation performs the matching process to search for the call path that has generated the sequence of logs. The result of the matching process is as follows:

```
Call Path Matching: Closest Matched Call Path

Level 4 - Testing.X:x12()

Level 3 - Testing.X:x2()

Level 2 - Testing.X:x32()

Level 1 - Testing.X:x43()
Level 0 - org.apache.log4j.Logger:error(java.lang.Object)

Level 5 - Testing.X:x12()

Level 4 - Testing.X:x13()

Level 3 - Testing.X:x2()

Level 2 - Testing.X:x32()

Level 1 - Testing.X:x43()

Level 0 - org.apache.log4j.Logger:error(java.lang.Object)

Level 5 - Testing.X:x12()

Level 4 - Testing.X:x2()

Level 3 - Testing.X:x32()

Level 2 - Testing.X:x33()

Level 1 - Testing.X:x43()

Level 0 - org.apache.log4j.Logger:error(java.lang.Object)

Level 6 - Testing.X:x12()

Level 5 - Testing.X:x13()

Level 4 - Testing.X:x2()

Level 3 - Testing.X:x32()

Level 2 - Testing.X:x33()

Level 1 - Testing.X:x43()

Level 0 - org.apache.log4j.Logger:error(java.lang.Object)
```

The correctness of the result can be verified with the dependency graph in **Figure 7.4**. Even though node `X:x13()` and node `X:x33()` do not generate a log at all, we still include them in the potential program call paths; as a result, there are four matched program call paths in total.

7.4 Sequential Pattern Searching in Historical Logs

We describe a process of searching for sequential patterns in historical logs. First we explain why it is necessary and helpful to add this process to the overall log analysis. Then we break down the process into several steps and describe each step in detail. The most important step is analyzing samples of logs in search of sequential patterns.

Logs recorded in log files reflect the program executions at runtime; hence they mostly appear in chronological order. We are interested in a set of adjacent logs that frequently appear in a specific sequential order over time, which are considered as a sequential pattern. We believe that such a sequential pattern not only represents the particular execution sequence of methods in the program, but also reflects the potential interrelations between the runtime events recorded by the logs; therefore an important part of the log analysis is to explore any frequent occurrences of sequential logs in history.

7.4.1 Collecting Samples of Sequential Logs

We presented the overall process of parsing logs in log files, extracting information from logs and storing the information into corresponding columns of a database table. Using the database as the main depository of logs makes it fairly convenient and efficient to collect samples of logs with the help of database SQL queries.

The SQL queries allow us to conveniently search for a certain number of logs for different kinds of objectives. For example, we can search for logs generated within a particular time period and/or on a particular server, we can search for logs with a particular status (code), we can search for logs generated from a particular source code file, we can even search for logs whose log messages contains a particular constant string and so on. This is a huge advantage that the database holds over plain-text file and XML representation in terms of data repository.

The error log is always one of the most important objectives when we explore the historical logs. More precisely, we are interested in the sequence of logs involving an error log. The sequence of logs records an error that occurred at least once during runtime in the past and reveals the serial program executions that likely led to the occurrence of the error. We search for such a sequence of logs in historical log files and collect them to form a knowledge base of past system errors. This knowledge base not only helps us understand the errors that have occurred in the past, but also helps us interpret the same error if it occurs in the future.

The process of collecting such a sequence of logs is straightforward. We identify all the error logs in the database table of historical logs. For each error log, we trace *backwards* a certain number of logs, such as five, and form a sample of sequential logs including the error log. Then we analyze this sample of sequential logs for a potential pattern.

7.4.2 Analyzing Samples of Sequential Logs for Patterns

We developed a class called `DataMining` whose method `sequentialPatternSearching` analyzes the samples of sequential logs for potential sequential patterns.

The analysis process is similar to the process of matching a sequence of logs with eligible program call paths. For each sample of sequential logs, the method matches the error log with the corresponding logging point and builds up the relevant program call path starting from the caller method of the matched logging point. Then it matches the logs in the sequence sample with the corresponding graph nodes when traversing the whole program call path.

The matched program call path and the sample of sequential logs form the sequential pattern that we are searching for. We maintain all the information related to this sequential pattern in memory during the analysis process and save it in a database table afterwards.

It is worth mentioning that, since we are interested in the sequential patterns that involve error logs from the past, it is likely that the system errors recorded by the error logs have already been investigated and fixed by support engineers before we identify them. The knowledge base of sequential patterns should allow the support engineers to enter their thoughts and investigation results of the system errors, so that the sequential patterns can include such complementary information for future reference.

7.4.3 Storing Sequential Patterns

Storing in Memory

We developed a class called `LogPattern` to store information about a sequential pattern in memory. The `LogPattern` class has the following members:

```
public class LogPattern {
    int pid;
    int startLogMid;
    int startLogLid;
    String startLogStatus;
    ArrayList<Integer> logSequenceMid;
    ArrayList<Integer> logSequenceLid;
    String selectCallPath;
    String matchedCallPath;
}
```

- `pid` - an integer representing a unique id assigned to each sequential pattern
- `startLogMid`, `startLogLid` and `startLogStatus` - representing information about the starting log of the sequential pattern in a *backward* direction
- `logSequenceMid` - an array list representing the ids of logs included in the sequential pattern
- `logSequenceLid` - an array list representing the ids of logging points that match with logs included in the sequential pattern
- `selectCallPath` - a string representing a set of program call paths that has the start log as the root node and includes all possible program call paths from the root node
- `matchedCallPath` - a string representing the program call path that matches with all the logs included in the sequential pattern

Saving into the Database

For storing sequential patterns, we prepare a database table called `sequential_pattern` with nine columns. `PID` stores a unique id assigned to each sequential pattern.

`StartLogMID`, `StartLogLID` and `StartLogStatus` store information about the starting log of the sequential pattern. `LogSequenceMID` stores the ids of logs included in the sequential pattern. `LogSequenceLID` stores the ids of logging points that match with logs included in the sequential pattern. `SelectCallPath` stores a string representation of program call paths that have the start log as the root node and include all possible program call paths from the root node. `MatchedCallPath` stores a string representation of the program call path that matches with all the logs included in the sequential pattern. The `PID` is a primary key. For the purpose of efficient data query, we have the table indexed with `StartLogLID`. The table is populated with data from the instance of class `LogPattern` in memory.

7.5 Matching Logs with Sequential Patterns

We are ready to describe the second of two different approaches to analyzing a sequence of logs. This approach attempts to match a sequence of logs with existing sequential patterns collected from historical logs, and searches for the sequential pattern that represents the sequence of logs and reflects the interrelations between program executions recorded by the logs.

7.5.1 General Process

The process of the second approach is simpler than the first approach, because most of the essential analysis work has been carried out and all necessary information is maintained in database tables. The remaining task of this approach is to search among existing sequential patterns for the one that matches with the sequence of logs. The process basically goes over the existing sequential patterns in the database and attempts to match each of them with the sequence of logs. The only matching criterion is the sequences of logging point ids from both sides. We consider them matched as long as either sequence is a subsequence of the other.

If a match is found, then all the information about the matched sequential pattern including the program call path can be obtained from the database. This information should be able to provide a solid understanding of the program executions at runtime that lead to the sequence of logs. Furthermore, if the sequence of logs involves an error log and the corresponding sequential pattern has more information about the same error from the past, as previously discussed, the complementary information can be helpful to support engineers investigating the present system error.

7.5.2 Example of Matching Logs with Sequential Patterns

We reconsider the sample program with the dependency graph as demonstrated in **Figure 7.3**. We have the same sequence of logs involving an error log:

```
DEBUG 2011-11-11 09:17:20,941 [main] Testing.X - x12

DEBUG 2011-11-11 09:17:21,956 [main] Testing.X - x2

DEBUG 2011-11-11 09:17:22,972 [main] Testing.X - x32

DEBUG 2011-11-11 09:17:23,988 [main] Testing.X - x42

ERROR 2011-11-11 09:17:24,050 [main] Testing.X - system error!
```

The result of the matching process is:

```
Sequential Pattern Matching: Closest Matched Sequential Pattern

...\testing.log - Line 11 - DEBUG 2011-10-10 08:19:30,347 [main] Testing.X - x12
```

```
...\testing.log - Line 12 - DEBUG 2011-10-10 08:19:31,586 [main] Testing.X - x2
...\testing.log - Line 13 - DEBUG 2011-10-10 08:19:32,450 [main] Testing.X - x32
...\testing.log - Line 14 - DEBUG 2011-10-10 08:19:33,124 [main] Testing.X - x42
...\testing.log - Line 15 - ERROR 2011-10-10 08:19:34,238 [main] Testing.X - syste

Sequential Pattern Matching: Possible Call Path(s)
Level 5 - Testing.X:main(java.lang.String[])
Level 4 - Testing.X:x11()
Level 5 - Testing.X:main(java.lang.String[])
Level 4 - Testing.X:x12()
Level 3 - Testing.X:x2()
Level 2 - Testing.X:x31()
Level 5 - Testing.X:main(java.lang.String[])
Level 4 - Testing.X:x11()
Level 5 - Testing.X:main(java.lang.String[])
Level 4 - Testing.X:x12()
Level 3 - Testing.X:x2()
Level 2 - Testing.X:x32()
Level 1 - Testing.X:x42()
Level 0 - org.apache.log4j.Logger:error(java.lang.Object)

Sequential Pattern Matching: Closest Matched Call Path
Level 4 - Testing.X:x12()
Level 3 - Testing.X:x2()
Level 2 - Testing.X:x31()

Level 1 - Testing.X:x42()

Level 0 - org.apache.log4j.Logger:error(java.lang.Object)
```

The correctness of the result can be verified with the dependency graph in **Figure 7.3**. The reader might realize that the results from the first and second approaches are identical, which demonstrates that both approaches are equally valid in terms of correctness.

7.6 Summary

In this chapter, we have presented implementation details of log analysis, which is the second of three major components in our log analysis project. We described the process of parsing logs in log files, extracting information from logs and matching each log with the corresponding logging point. We also described the process of building the access dependency graph from the access dependency relations and searching for sequential patterns in historical logs.

The most important subject matter in this chapter is the general process of analyzing a sequence of logs by applying various information we have collected. The process includes two different approaches: matching the sequence of logs with program call paths and matching the sequence of logs with existing sequential patterns. The log analysis process described so far has demonstrated that the methodology and system design of our project is capable of providing a feasible solution to the problem of diagnosis of software defects.

In the next chapter, we will present the last major component in the log analysis project, i.e., sequential pattern mining. We will discuss how the application of sequential pattern mining helps refine the results of log analysis.

Chapter 8

Sequential Pattern Mining

In this chapter, we present the last of three major components, i.e., sequential pattern mining, as shown in **Figure 8.1**. We revisit the rationale behind the decision to apply the sequential pattern mining technique in the log analysis process. We want to demonstrate that, by mining potential sequential patterns hidden in the log analysis results, this technique can provide an extended insight into the software system from a statistical point of view. In order to support our argument, we will provide an example of mining sequential patterns. We run the mining algorithm on some sample data, explain the output of the mining process and discuss the implications for our log analysis project.

Rationale

Sequential pattern mining discovers frequent subsequences as patterns in a sequence database. By applying the sequential pattern mining technique in the log analysis process, we intend to explore potential sequential patterns among historical logs, reveal interrelations between events recorded by logs and further study the corresponding

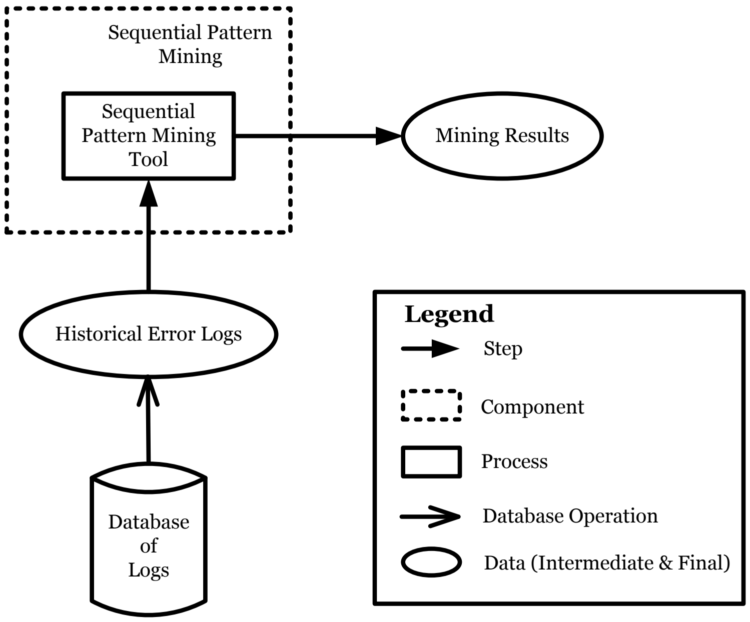


Figure 8.1: Process of Sequential Pattern Mining

sequences of program executions behind these events; therefore the sequential pattern mining is considered as a complementary approach to the log analysis process. It provides helpful information about the software system in the past and assists support engineers in diagnosing system errors at present and in the future.

8.1 The Implementation

There has been extensive research and development work in the field of sequential pattern mining, in both the industrial and academic domains. Software researchers and engineers have implemented the advanced mining methods and algorithms into practical applications, so that the mining technique can be broadly utilized in a larger number of industries.

Fournier-Viger [FV12] has developed an open-source data mining framework written in Java. It was originally a sequential pattern mining framework; hence it was named as SPMF. The SPMF framework now offers implementations of many different data mining algorithms for discovering sequential patterns, sequential rules, association rules, frequent itemsets and more.

We adopted Fournier-Viger's implementation of the PrefixSpan algorithm for mining frequent sequential patterns from a sequence database, which is researched and developed by Pei *et al.* [PHMA+04]. PrefixSpan mines the complete set of patterns but greatly reduces the effort of candidate subsequence generation. A characteristic of this algorithm is that it explores prefix-projection in sequential pattern mining. The prefix-projection substantially reduces the size of projected databases and leads to more efficient processing when compared to other sequential pattern mining methods.

8.2 Example of Sequential Pattern Mining

General Process

We have successfully integrated Fournier-Viger's implementation of the PrefixSpan algorithm [FV12] into our project. The sequential pattern mining process takes a set of log sequences as input and applies a mining algorithm to identify potential sequential patterns in these log sequences.

Suppose that we have collected a set of log sequences from historical logs in the database through log analysis. The log in each sequence is represented by the id of the corresponding logging point. Each sequence involves an error log represented by the last logging point id of the sequence. These sequences of logs actually reflect different program call paths that have caused system errors in the past. The order of program execution is from left to right in each sequence.

```
0: (19 21 22 24 25 )
1: (19 21 22 26 27 )
2: (19 21 23 26 27 )
3: (19 21 23 28 29 )
4: (20 21 22 24 25 )
5: (20 21 22 26 27 )
6: (20 21 23 26 27 )
7: (20 21 23 28 29 )
```

The reader might observe that some sequences share a certain number of logging points in common. It shows that these sequences of logs are not completely independent cases, instead, there are potential interrelations between the logging points or the caller methods of the logging points hidden in the sequences. We attempt to explore such interrelations with the help of sequential pattern mining technique.

We have a *sequence database* consisted of all the *sequences* presented. The set of *items* in the database is $\{19, 21, 22, 23, 24, 25, 26, 27, 28, 29\}$. The basic task of sequential pattern mining is to find all sequential patterns that occur in more than the *min_support* number of sequences of the database. The *min_support* is a percentage or positive integer representing the *support threshold* for a *sequential pattern*. A *subsequence* is considered as a *sequential pattern* in the *sequence database* only when the total number of occurrences of the *subsequence* is no less than the value of *min_support*.

We ran Fournier-Viger's implementation of the PrefixSpan algorithm [FV12] with *min_support* as 50%, which means a *subsequence* has to appear in at least half of the sequences of the database in order to be considered as a *sequential pattern*. We have the mining result as follows:

```
--------- FREQUENT SEQUENTIAL PATTERNS ---------
L0
L1
pattern 1:  (19 )       Sequence ID: 0 1 2 3         support:  0.5 (4/8)
pattern 2:  (19 21 )    Sequence ID: 0 1 2 3         support:  0.5 (4/8)
pattern 3:  (21 )       Sequence ID: 0 1 2 3 4 5 6 7 support:  1   (8/8)
pattern 4:  (21 27 )    Sequence ID: 1 2 5 6         support:  0.5 (4/8)
pattern 5:  (21 26 )    Sequence ID: 1 2 5 6         support:  0.5 (4/8)
pattern 6:  (21 26 27 ) Sequence ID: 1 2 5 6         support:  0.5 (4/8)
pattern 7:  (21 23 )    Sequence ID: 2 3 6 7         support:  0.5 (4/8)
pattern 8:  (21 22 )    Sequence ID: 0 1 4 5         support:  0.5 (4/8)
pattern 9:  (20 )       Sequence ID: 4 5 6 7         support:  0.5 (4/8)
pattern 10: (20 21 )    Sequence ID: 4 5 6 7         support:  0.5 (4/8)
pattern 11: (23 )       Sequence ID: 2 3 6 7         support:  0.5 (4/8)
pattern 12: (22 )       Sequence ID: 0 1 4 5         support:  0.5 (4/8)
pattern 13: (27 )       Sequence ID: 1 2 5 6         support:  0.5 (4/8)
pattern 14: (26 )       Sequence ID: 1 2 5 6         support:  0.5 (4/8)
pattern 15: (26 27 )    Sequence ID: 1 2 5 6         support:  0.5 (4/8)
```

We observe the sequential pattern mining result and focus on the extreme or unique characteristics of sequential patterns that have been identified by the mining algorithm. We interpret the implications of these findings for more insight into the software system.

Observations

- There are 15 sequential patterns in total that appear in at least half of the sequences of the database.
- The sequential pattern with most occurrences is `pattern 3`, whose item, `21`, appears in all the sequences of the database.
- The longest sequential pattern is `pattern 6`, which consists of 3 items: `21`, `26` and `27`.
- Even though `pattern 6` is the longest sequential pattern, it is not the longest consecutive sequential pattern. Some of the items that constitute the pattern are not adjacent to each other in some sequences. For example, item `26` is not adjacent to item `21` in any sequence at all.
- The longest consecutive sequential pattern has 2 items. There are 5 such patterns in total: `pattern 2`, `pattern 7`, `pattern 8`, `pattern 10` and `pattern 15`.
- The items of `25`, `27` and `29` represent the logging points that generated the error logs. Only item `27` appears in at least half of the sequences of the database and is involved in some sequential patterns.
- Item `19` starts half of the sequences of the database and item `20` starts the other half. Both of these items represent the logging points and the caller

methods of the logging points that started the program executions resulting in the corresponding sequences of logs.

Implications

- The sequential patterns with most occurrences imply the logging point or the caller method of the logging point that was most frequently involved in the system errors.
- The longest sequential pattern implies the longest subsequence of items that is included in half of the sequences in the database. It expands the range of logging points or caller methods of the logging points that were frequently involved in the system errors.
- The longest consecutive sequential pattern implies the serial method calls that have most frequently caused the eventual system errors.
- The item that represents the logging point for the error log and has the most occurrences implies that the caller method of the logging point was the method where the system errors most frequently occurred.
- Similarly, the item that starts the highest number of sequences in the database implies that the caller method of the corresponding logging point started the highest number of program executions that led to eventual system errors.

Conclusion

We have applied the sequential pattern mining algorithm to a set of log sequences through the integrated mining implementation. The algorithm explores different sections of logs sequences and determines the potential sequential patterns among them. Through the observation of mining results, we are able to identify certain extreme or unique characteristics of the sequential patterns, such as the sequential patterns with the most occurrences, the longest (consecutive) sequential pattern, and the special items in these sequential patterns. The implications of these findings not only reflect the interrelations between the log entries that make up of the patterns, but also provide extended insight into the software system, such as the program call paths that are matched with the log sequences and the methods that comprise the program call paths.

The mining algorithm also calculates the frequencies of all identified sequential patterns. From the statistical point of view, the frequency of an event or pattern in the past calculated by the mining algorithm actually implies the recurrence probability of the same event or pattern in the future; therefore, even though we have made the implications based on the sequential patterns of historical data, the implications not only explain the events that occurred in the past, but also help us interpret any similar events that may occur in the future.

8.3 Summary

In this chapter, we have presented the sequential pattern mining process, which is the last of three major components in our log analysis project. We performed the mining technique on some sample sequences of logs and made some observations of the sequential patterns identified by the mining algorithm. We discussed the implications of these findings, in order to demonstrate that, by mining potential sequential patterns hidden in the log analysis results, this technique can provide an extended insight into the software system from a statistical point of view.

In the next chapter, we will evaluate the implementation of the log analysis project with experimental results. We deploy the implementation to our partner's legacy system, analyze sample logs from the system and verify the effectiveness of analysis results.

Chapter 9

Evaluation

In this chapter, we evaluate our implementation of the log analysis project. The intention of the log analysis project presented in this thesis is to provide a feasible solution to automatic diagnosis of software defects, which is also our partner's primary concern. We deploy our implementation on the partner's software system, conduct experiments on analyzing sample logs from the system and verify the effectiveness of analysis results.

9.1 Our Partner's System

Our partner maintains a legacy software system that was developed years ago. The core of the system was implemented in Java programming language; therefore the compiled binaries of source code, i.e., Java bytecode (*.class*) files, are available for source code analysis.

Logs generated by the system at runtime are maintained in plain-text files. The logging framework used in the system is Apache log4j, but there are customized logging methods implemented by developers, which generate logs in a slightly different format from the ones generated by log4j methods. Even though it needs to be advised about the particular log format, all the logs can be parsed conveniently by the log parser of our implementation.

Error logs are recorded in log files as well, which are marked with an error status code. On the occasions of system failures, support engineers of the software system follow the process of conventional log analysis to investigate the corresponding error logs for the root cause. Some identified error logs, whose root causes are already known, are used in our experiments for the purpose of evaluation.

9.2 Evaluation Approach

For the purpose of evaluation, we collected over 6 million lines of logs from our partner's system, extracted execution information from each log, searched for sequential patterns of logs related to system errors in the past and summarized essential information for automatic diagnosis. We also performed source code analysis on the Java bytecode files, analyzed dependency relations among entities and stored collected information into database tables for eventual log analysis.

We have presented the design and implementation details of our solution to support engineers of our partner's software system, so that they understand the general process of log analysis. One of the support engineers has selected several sequences of logs for case studies to test the effectiveness of our implementation. These sequences of logs have been manually investigated by support engineers and the corresponding information on program executions, such as program call paths, are already identified. Support engineers manually examined the results generated by automatic log analysis. They compared the analysis results against the manual diagnosis, in order to verify the effectiveness of analysis results.

If our implementation infers a set of program call paths potentially matching with a particular sequence of logs involving an error log, we consider the result *useful*. If our implementation infers one and only one program call path correctly matching with the particular sequence of logs involving an error log, we consider the result *complete*.

We mainly focus on verifying the results of log analysis. Since the results of log analysis are built upon information collected by source code analysis, log parsing and sequential pattern searching, the effectiveness of all these processes can be examined by the verification of the eventual results of log analysis.

Our experiments were conducted on a Windows XP machine with Intel Core 2 2.13GHz CPU and 2GB of memory. Our implementation is a single threaded program. We analyze one sequence of sample logs at a time and collect the corresponding diagnostic information.

9.3 Experimental Results

Our implementation provides two different approaches to analyzing a sequence of logs: matching the sequence of logs with potential program call paths and matching the sequence of logs with existing sequential patterns of logs reflecting historical system errors. The first three case studies demonstrate the effectiveness of these two approaches in finding the corresponding program call path for the log sequence.

The sequential pattern mining is another important component of our implementation. The last case demonstrates that, by applying sequential pattern mining technique in the log analysis process, we are able to explore potential sequential patterns among historical logs, reveal interrelations between events recorded by logs and provide an extended insight of our partner's system from a statistical point of view.

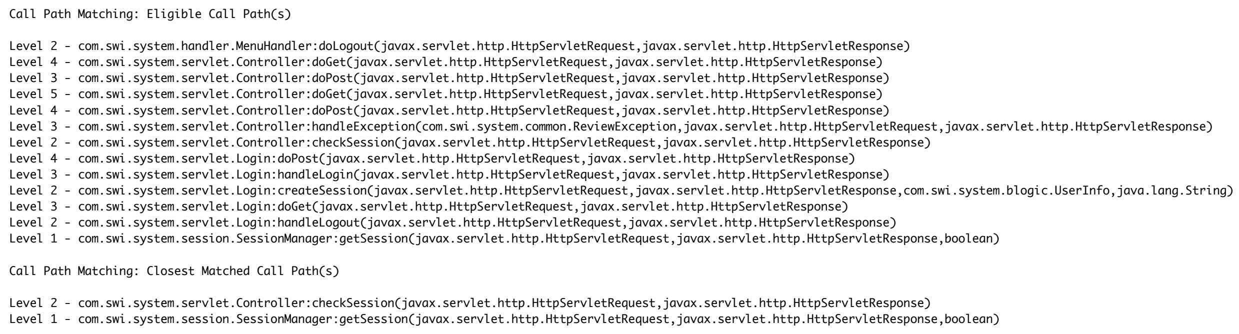
9.3.1 A Case of Matching Error Logs with Program Call Paths

We have a sequence of sample logs indicating an exception error as shown in **Figure 9.1**.

```
DEBUG 2011-10-28 19:21:03,511 [[ACTIVE] ExecuteThread] com.swi.system.session.SessionManager - getLastSession
DEBUG 2011-10-28 19:21:03,511 [[ACTIVE] ExecuteThread] com.swi.system.servlet.Controller - checkSession
DEBUG 2011-10-28 19:21:03,511 [[ACTIVE] ExecuteThread] com.swi.system.session.SessionManager - getSession
ERROR 2011-10-28 19:21:03,511 [[ACTIVE] ExecuteThread] ReviewException.INTERNAL_ERROR - system:Internal Error
weblogic.utils.NestedRuntimeException: Cannot parse POST parameters of request: '/controller'
        at weblogic.servlet.internal.ServletRequestImpl$RequestParameters.mergePostParams(ServletRequestImpl.java:1812)
        at weblogic.servlet.internal.ServletRequestImpl$RequestParameters.parseQueryParams(ServletRequestImpl.java:1699)
        at weblogic.servlet.internal.ServletRequestImpl$RequestParameters.getQueryParams(ServletRequestImpl.java:1652)
        at weblogic.servlet.internal.ServletRequestImpl.getParameterNames(ServletRequestImpl.java:756)
        at com.swi.system.servlet.AppServletRequest.<init>(AppServletRequest.java:67)
        at com.swi.system.servlet.Controller.doPost(Controller.java:352)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:763)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
        at weblogic.servlet.internal.StubSecurityHelper$ServletServiceAction.run(StubSecurityHelper.java:227)
        at weblogic.servlet.internal.StubSecurityHelper.invokeServlet(StubSecurityHelper.java:125)
        at weblogic.servlet.internal.ServletStubImpl.execute(ServletStubImpl.java:283)
        at weblogic.servlet.internal.ServletStubImpl.execute(ServletStubImpl.java:175)
        at weblogic.servlet.internal.WebAppServletContext$ServletInvocationAction.run(WebAppServletContext.java:3228)
        at weblogic.security.acl.internal.AuthenticatedSubject.doAs(AuthenticatedSubject.java:321)
        at weblogic.security.service.SecurityManager.runAs(SecurityManager.java:121)
        at weblogic.servlet.internal.WebAppServletContext.securedExecute(WebAppServletContext.java:2002)
        at weblogic.servlet.internal.WebAppServletContext.execute(WebAppServletContext.java:1908)
        at weblogic.servlet.internal.ServletRequestImpl.run(ServletRequestImpl.java:1362)
        at weblogic.work.ExecuteThread.execute(ExecuteThread.java:209)
        at weblogic.work.ExecuteThread.run(ExecuteThread.java:181)
Caused by: java.net.SocketException: Connection reset
        at java.net.SocketInputStream.read(SocketInputStream.java:168)
        at weblogic.utils.io.ChunkedInputStream.read(ChunkedInputStream.java:159)
        at java.io.InputStream.read(InputStream.java:89)
        at com.certicom.tls.record.ReadHandler.readFragment(Unknown Source)
        at com.certicom.tls.record.ReadHandler.readRecord(Unknown Source)
        at com.certicom.tls.record.ReadHandler.read(Unknown Source)
        at com.certicom.io.InputSSLIOStreamWrapper.read(Unknown Source)
        at weblogic.servlet.internal.PostInputStream.read(PostInputStream.java:177)
        at weblogic.servlet.internal.ServletInputStreamImpl.read(ServletInputStreamImpl.java:211)
        at weblogic.servlet.internal.ServletRequestImpl$RequestParameters.mergePostParams(ServletRequestImpl.java:1787)
        ... 19 more
```

Figure 9.1: Sample Error Logs

The result generated from log analysis with the approach of matching program call paths is as shown in **Figure 9.2**. It lists all of the program call paths that could potentially lead to the indicated exception error. The list is generated from the



```
Call Path Matching: Eligible Call Path(s)

Level 2 - com.swi.system.handler.MenuHandler:doLogout(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Controller:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 5 - com.swi.system.servlet.Controller:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Controller:handleException(com.swi.system.common.ReviewException,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Controller:checkSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Login:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Login:handleLogin(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Login:createSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,com.swi.system.blogic.UserInfo,java.lang.String)
Level 3 - com.swi.system.servlet.Login:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Login:handleLogout(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 1 - com.swi.system.session.SessionManager:getSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,boolean)

Call Path Matching: Closest Matched Call Path(s)

Level 2 - com.swi.system.servlet.Controller:checkSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 1 - com.swi.system.session.SessionManager:getSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,boolean)
```

Figure 9.2: Result of Matching Sample Error Logs with Program Call Paths

initial analysis, which traverses the dependency graph with a depth-first search (DFS) algorithm starting from the method that generated the error log. There are five in total of such program call paths; however, after the final analysis, it is determined that only one program call path matches with the sequence of logs. It has been verified that the result is correct and exact, which means the only matched program call path correctly shows the methods related to the cause of the exception error.

The result also reveals that, even though the sample log sequence consists of four log entries, it doesn't necessarily mean all of the four log entries are generated by the execution of one program call path. In this case, the first line of the log entry with message contents of "*getLastSession*" is not generated by the matched program call path.

9.3.2 A Case of Matching Normal Logs with Program Call Paths

The first case represents the typical process of analyzing error logs on the occasion of system errors. Besides error logs, our implementation can analyze any sequence of non-error logs and reveal the program call path that possibly generated the particular sequence.

We have another sequence of sample logs as shown in **Figure 9.3**.

```
INFO  2012-03-14 09:27:41,326 [[ACTIVE] ExecuteThread] com.swi.system.servlet.Controller - Redirecting to jsp page
DEBUG 2012-03-14 09:27:41,326 [[ACTIVE] ExecuteThread] com.swi.system.servlet.Util - Util.forward
```

Figure 9.3: Sample Normal Logs

```
Call Path Matching: Eligible Call Path(s)

Level 2 - com.swi.system.filter.Redirect:doFilter(javax.servlet.ServletRequest,javax.servlet.ServletResponse,javax.servlet.FilterChain)
Level 3 - com.swi.system.servlet.Controller:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Controller:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Controller:handleException(com.swi.system.common.ReviewException,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Controller:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Controller:handleMessage(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,java.lang.String,int)
Level 2 - com.swi.system.servlet.Controller:handleMessage(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,java.lang.String,java.lang.String,java.lang.String)
Level 2 - com.swi.system.servlet.Login:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Login:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Login:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Login:handleForgotOne(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Login:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Login:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Util:showException(com.swi.system.common.ReviewException,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Util:showException(com.swi.system.common.ReviewException,java.lang.String,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 5 - com.swi.system.servlet.Login:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 5 - com.swi.system.servlet.Login:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Util:showException(com.swi.system.common.ReviewException,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Util:showException(com.swi.system.common.ReviewException,java.lang.String,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Util:showResult(java.lang.String,java.lang.String,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 1 - com.swi.system.servlet.Util:forward(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,java.lang.String)

Call Path Matching: Closest Matched Call Path(s)

Level 2 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 1 - com.swi.system.servlet.Util:forward(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,java.lang.String)
Level 3 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Controller:handleException(com.swi.system.common.ReviewException,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 1 - com.swi.system.servlet.Util:forward(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,java.lang.String)
Level 3 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Controller:handleMessage(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,java.lang.String,int)
Level 1 - com.swi.system.servlet.Util:forward(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,java.lang.String)
```

Figure 9.4: Result of Matching Sample Normal Logs with Program Call Paths

The result generated from log analysis is as shown in **Figure 9.4**. Even though there are only two logs in the sequence, there are twelve program call paths that may lead to the recorded program execution. The range is narrowed down to three after the final analysis. It has been verified that the result is correct, but not exact.

The reason why the result is not exact is that methods can be executed without generating logs. This completely depends on the design of the methods. Some methods are not programmed to generate a log at all, which means there is no logging point in the method; some methods have logging points, but due to the restrictions of conditional statements, such as *if-else* statements, none of the logging points is executed and as a result, no log is generated. Either of these two scenarios could have led to the result we have in this case.

The reader might recognize from the result that all of the three program call paths start with the same entity (method) `com.swi.system.servlet.Util:forward()`, which matches with the second line of log in the sequence, and all of the three paths end with the same entity (method) `com.swi.system.servlet.Controller:doPost()`, which matches with the first line of log in the sequence. Since all of the three program call paths match the start and end log of the sequence, we consider all of them as most closely matched.

9.3.3 A Case of Matching Error Logs with Sequential Patterns

We will reconsider the sequence of sample logs indicating an exception error as shown in **Figure 9.1**. We intend to demonstrate that, besides the approach of matching logs with program call paths, another approach of matching logs with sequential patterns can also generated an effective result.

The result generated from log analysis with the approach of matching sequential patterns is as shown in **Figure 9.5**. We happened to identify a pattern match between the sequence of error logs and an existing log sequence in the database indicating a similar exception error. Since the exception error has already been identified and investigated, the previous analysis result was available in the database for retrieval. The result includes all pieces of information about the matched sequential pattern. It lists all of the logs making up of the pattern. It also lists all of the program call

```
Sequential Pattern Matching: Closest Matched Sequential Pattern

C:\ev\logs\yellow1\system.log.68 - Line 15639 - DEBUG 2011-10-27 22:55:39,007 [[ACTIVE] ExecuteThread] com.swi.system.database.DbBase - Database connection returned to connection pool
C:\ev\logs\yellow1\system.log.68 - Line 15640 - DEBUG 2011-10-27 22:55:39,022 [[ACTIVE] ExecuteThread] com.swi.system.session.SessionManager - getLastSession
C:\ev\logs\yellow1\system.log.68 - Line 15641 - DEBUG 2011-10-27 22:55:39,022 [[ACTIVE] ExecuteThread] com.swi.system.servlet.Controller - checkSession
C:\ev\logs\yellow1\system.log.68 - Line 15642 - DEBUG 2011-10-27 22:55:39,022 [[ACTIVE] ExecuteThread] com.swi.system.session.SessionManager - getSession
C:\ev\logs\yellow1\system.log.68 - Line 15643 - ERROR 2011-10-27 22:55:39,022 [[ACTIVE] ExecuteThread] ReviewException.INTERNAL_ERROR - system:Internal Error
weblogic.utils.NestedRuntimeException: Cannot parse POST parameters of request: '/controller'
        at weblogic.servlet.internal.ServletRequestImpl$RequestParameters.mergePostParams(ServletRequestImpl.java:1812)
        at weblogic.servlet.internal.ServletRequestImpl$RequestParameters.parseQueryParams(ServletRequestImpl.java:1699)
        at weblogic.servlet.internal.ServletRequestImpl$RequestParameters.getQueryParams(ServletRequestImpl.java:1652)
        at weblogic.servlet.internal.ServletRequestImpl.getParameterNames(ServletRequestImpl.java:756)
        at com.swi.system.servlet.AppServletRequest.<init>(AppServletRequest.java:67)
        at com.swi.system.servlet.Controller.doPost(Controller.java:352)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:763)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
        at weblogic.servlet.internal.StubSecurityHelper$ServletServiceAction.run(StubSecurityHelper.java:227)
        at weblogic.servlet.internal.StubSecurityHelper.invokeServlet(StubSecurityHelper.java:125)
        at weblogic.servlet.internal.ServletStubImpl.execute(ServletStubImpl.java:283)
        at weblogic.servlet.internal.ServletStubImpl.execute(ServletStubImpl.java:175)
        at weblogic.servlet.internal.WebAppServletContext$ServletInvocationAction.run(WebAppServletContext.java:3228)
        at weblogic.security.acl.internal.AuthenticatedSubject.doAs(AuthenticatedSubject.java:321)
        at weblogic.security.service.SecurityManager.runAs(SecurityManager.java:121)
        at weblogic.servlet.internal.WebAppServletContext.securedExecute(WebAppServletContext.java:2002)
        at weblogic.servlet.internal.WebAppServletContext.execute(WebAppServletContext.java:1908)
        at weblogic.servlet.internal.ServletRequestImpl.run(ServletRequestImpl.java:1362)
        at weblogic.work.ExecuteThread.execute(ExecuteThread.java:209)
        at weblogic.work.ExecuteThread.run(ExecuteThread.java:181)
Caused by: java.net.SocketException: Connection reset
        at java.net.SocketInputStream.read(SocketInputStream.java:168)
        at weblogic.utils.io.ChunkedInputStream.read(ChunkedInputStream.java:159)
        at java.io.InputStream.read(InputStream.java:89)
        at com.certicom.tls.record.ReadHandler.readFragment(Unknown Source)
        at com.certicom.tls.record.ReadHandler.readRecord(Unknown Source)
        at com.certicom.tls.record.ReadHandler.read(Unknown Source)
        at com.certicom.io.InputSSLIOStreamWrapper.read(Unknown Source)
        at weblogic.servlet.internal.PostInputStream.read(PostInputStream.java:177)
        at weblogic.servlet.internal.ServletInputStreamImpl.read(ServletInputStreamImpl.java:211)
        at weblogic.servlet.internal.ServletRequestImpl$RequestParameters.mergePostParams(ServletRequestImpl.java:1787)
        ... 19 more

Sequential Pattern Matching: Eligible Call Path(s)

Level 2 - com.swi.system.handler.MenuHandler:doLogout(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Controller:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 5 - com.swi.system.servlet.Controller:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Controller:handleException(com.swi.system.common.ReviewException,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Controller:checkSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 4 - com.swi.system.servlet.Login:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 3 - com.swi.system.servlet.Login:handleLogin(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Login:createSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,com.swi.system.blogic.UserInfo,java.lang.String)
Level 3 - com.swi.system.servlet.Login:doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 2 - com.swi.system.servlet.Login:handleLogout(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 1 - com.swi.system.session.SessionManager:getSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,boolean)

Sequential Pattern Matching: Closest Matched Call Path(s)

Level 2 - com.swi.system.servlet.Controller:checkSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Level 1 - com.swi.system.session.SessionManager:getSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,boolean)
```

Figure 9.5: Result of Matching Sample Logs with Sequential Patterns

paths that are most closely matched with the sequential pattern, which are also most closely matched with the sequence of sample logs.

The advantage of matching error logs with existing sequential patterns is efficiency. All previous analysis information is collected during the process of sequential pattern searching and maintained in a database table as a knowledge base. When a match between the new sequence of error logs and the existing sequential pattern is identified, the previous analysis information can be directly retrieved from the knowledge base without repeating the process of matching error logs with program call paths. It is much more efficient and still provides sufficient and accurate diagnostic information.

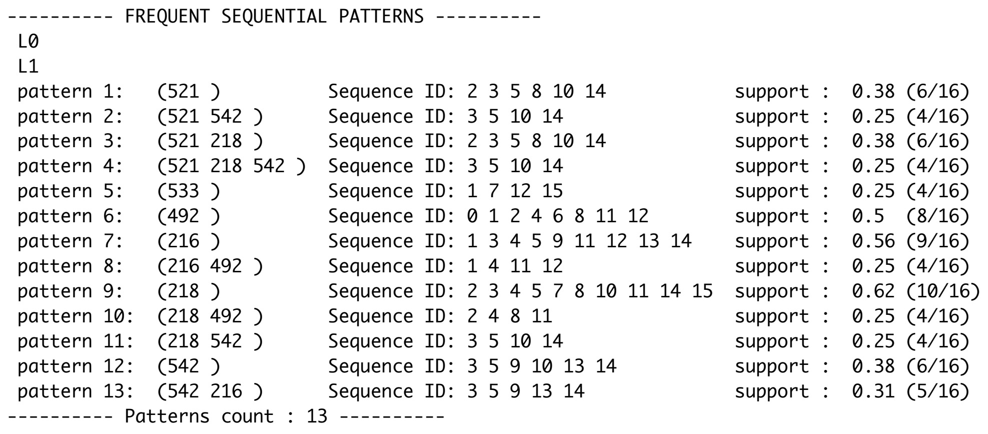
9.3.4 A Case of Sequential Pattern Mining

We collect different log sequences related to past system errors from a large number of historical logs. But these log sequences only reflect system errors in the past on an individual basis, there is no available information to reveal the interrelations between multiple sequences of error logs, if any exists. The sequential pattern mining technique helps us explore such kinds of relations between program executions that have led to system failures. It looks for sequential patterns in the log sequences that can provide an extended insight of our partner's system from a statistical point of view.

Each of the log sequences that we have collected are represented by the logging point ids of logs. The order of program execution of each log sequence is from left to right. All of the log sequences are organized as shown in **Figure 9.6** and used as input for sequential pattern mining.

```
0:  (518 490 517 492 )
1:  (519 533 216 492 )
2:  (218 521 218 492 )
3:  (521 218 542 216 )
4:  (458 216 218 492 )
5:  (521 218 542 216 )
6:  (517 520 484 492 )
7:  (218 531 480 533 )
8:  (218 521 218 492 )
9:  (518 491 542 216 )
10: (218 521 218 542 )
11: (458 216 218 492 )
12: (519 533 216 492 )
13: (489 542 216 537 )
14: (521 218 542 216 )
15: (218 531 480 533 )
```

Figure 9.6: The Sequences for Sequential Pattern Mining



```
---------- FREQUENT SEQUENTIAL PATTERNS ----------
 L0
 L1
 pattern 1:   (521 )          Sequence ID: 2 3 5 8 10 14          support :  0.38 (6/16)
 pattern 2:   (521 542 )      Sequence ID: 3 5 10 14              support :  0.25 (4/16)
 pattern 3:   (521 218 )      Sequence ID: 2 3 5 8 10 14          support :  0.38 (6/16)
 pattern 4:   (521 218 542 )  Sequence ID: 3 5 10 14              support :  0.25 (4/16)
 pattern 5:   (533 )          Sequence ID: 1 7 12 15              support :  0.25 (4/16)
 pattern 6:   (492 )          Sequence ID: 0 1 2 4 6 8 11 12      support :  0.5  (8/16)
 pattern 7:   (216 )          Sequence ID: 1 3 4 5 9 11 12 13 14  support :  0.56 (9/16)
 pattern 8:   (216 492 )      Sequence ID: 1 4 11 12              support :  0.25 (4/16)
 pattern 9:   (218 )          Sequence ID: 2 3 4 5 7 8 10 11 14 15 support :  0.62 (10/16)
 pattern 10:  (218 492 )      Sequence ID: 2 4 8 11               support :  0.25 (4/16)
 pattern 11:  (218 542 )      Sequence ID: 3 5 10 14              support :  0.25 (4/16)
 pattern 12:  (542 )          Sequence ID: 3 5 9 10 13 14         support :  0.38 (6/16)
 pattern 13:  (542 216 )      Sequence ID: 3 5 9 13 14            support :  0.31 (5/16)
---------- Patterns count : 13 ----------
```

Figure 9.7: Result of Sequential Pattern Mining

We have set the *support threshold* (*min_support*) for a *sequential pattern* as 25%, which means a *subsequence* has to appear in only a quarter of the sequences in order to be considered as a *sequential pattern*.

The result of sequential pattern mining is shown in **Figure 9.7**. There are two levels of mining processes, which generate thirteen sequential patterns in total.

Observations and Implications

- Most of the identified sequential patterns have supports between 25% and 50%. Only three patterns, i.e., `pattern 6`, `pattern 7`, `pattern 9`, have supports equal to or higher than 50%. This implies that the majority of sequential patterns don't re-appear very often.

- The sequential pattern with most occurrences is `pattern 9`, whose item `218` appears in 10 out of 16 sequences. The caller entity (method) of the logging point with id `218` is as shown in **Figure 9.8**. It is a database related method. This implies that database operations are probably involved in most system errors in the past.

```
LID - Entity
218 - com.swi.system.database.DbBase:destroy()
```

Figure 9.8: Entity of Pattern 9

- The longest sequential pattern is `pattern 4`, which consists of three items: `512`, `218` and `542`. `pattern 4` is also the longest consecutive sequential pattern. The three items are adjacent to each other in all of the four appearances. The caller entities (methods) of the three items are as shown in **Figure 9.9**. The longest consecutive sequential pattern implies the serial method executions that have most frequently caused eventual system errors. Furthermore, the longest sequential pattern contains the sequential pattern with most occurrences, i.e. item `218`, which again implies that database operations are involved in the majority of system errors in the past.

```
LID - Entity
521 - com.swi.system.session.SessionManager:checkForSessionTimeout()
218 - com.swi.system.database.DbBase:destroy()
542 - com.swi.system.session.SessionManager:getLatestSession(com.swi.system.session.Session,com.swi.system.session.Session)
```

Figure 9.9: Entities of Pattern 4

- The items `533`, `492`, `216`, `542` and `537` represent the logging points that were executed before the occurrence of system errors. Except item `537`, the rest of the items appear in 4 or more out of 16 (25%) sequences. The caller entities (methods) of these items are as shown in **Figure 9.10**. Even though the caller entities (methods) of logging points may not have directly caused the eventual system errors, they are the last methods that were executed and generated logs before the failed execution. They may record significant execution information that can be analyzed for the root cause and they are good start points for the log analysis process.

```
LID - Entity
533 - com.swi.system.session.SessionManager:getSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,boolean)

492 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)

216 - com.swi.system.database.DbBase DbBase(java.lang.String)

542 - com.swi.system.session.SessionManager getLatestSession(com.swi.system.session.Session,com.swi.system.session.Session)
```

Figure 9.10: Entities of Pattern 5, 6, 7 and 12

- There are seven sequential patterns which consist of more than one item: `pattern 2`, `pattern 3`, `pattern 4`, `pattern 8`, `pattern 10`, `pattern 11` and `pattern 13`. We have already interpreted `pattern 4`, which is the longest consecutive sequential pattern. The items of `pattern 3`, i.e., `521` and `542`, actually are not adjacent to each other in all of their appearances. The items of the rest sequential patterns mentioned are adjacent to each other in at least one of their appearances. The caller methods of the items in these sequential patterns are as shown in **Figure 9.11**. The sequential patterns consisting of consecutive items implies the sub-paths of program call paths in sequences. The adjacent items reveal the potential method calls during program execution. This is helpful information when we analyze the sequence of logs.

- Remember that we match a sequence of logs with an existing sequential pattern in Case 3. The matched sequential pattern consists of logging points with ids `(218 531 480 533 )`, which is one of the input sequences for the mining process. The caller methods of these logging points are as shown in **Figure 9.12**.

```
LID - Entity
521 - com.swi.system.session.SessionManager:checkForSessionTimeout()
218 - com.swi.system.database.DbBase:destroy()

216 - com.swi.system.database.DbBase:DbBase(java.lang.String)
492 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)

218 - com.swi.system.database.DbBase:destroy()
492 - com.swi.system.servlet.Controller:doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)

218 - com.swi.system.database.DbBase:destroy()
542 - com.swi.system.session.SessionManager getLatestSession(com.swi.system.session.Session,com.swi.system.session.Session)

542 - com.swi.system.session.SessionManager getLatestSession(com.swi.system.session.Session,com.swi.system.session.Session)
216 - com.swi.system.database.DbBase DbBase(java.lang.String)
```

Figure 9.11: Entities of Pattern 3, 8, 10, 11 and 13

```
LID - Entity
218 - com.swi.system.database.DbBase:destroy()
531 - com.swi.system.session.SessionManager:getLatestSession(com.swi.system.session.Session,com.swi.system.session.Session)
480 - com.swi.system.servlet.Controller:checkSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
533 - com.swi.system.session.SessionManager:getSession(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse,boolean)
```

Figure 9.12: Entities of Sequence (218 531 480 533)

We explore details of the mining result and focus on the extreme or unique characteristics of the sequential patterns identified by the mining algorithm. The extended information indeed provides us with more insight into system errors in the past and helps us better interpret and analyze any similar errors that may occur in the future.

9.3.5 Discussions of Experimental Results

We have applied two different approaches, matching errors logs with program call paths and matching error logs with sequential patterns, to analyze some selected sample sequences of logs. We have also applied the sequential pattern mining technique to provide extended information about certain system errors in the past from the statistical point of view. The results are summarized in **Table 9.1**.

Case	Descriptions	Analysis Approach	Result	Comments
1	• Analyzing a log sequence indicating a system error. • The log sequence consists of four log entries.	Matching Logs with Program Call Paths	Successful	• The initial analysis recognizes 5 potential program call paths that could lead to the indicated system error. • The final analysis determines that only 1 program call path matches the log sequence. • The result is correct and exact. • All of the potential program call paths are correct according to system design. • The only matched program call path reveals methods that were related to the cause of the system error.
2	• Analyzing a log sequence recording a normal program execution. • The log sequence consists of two log entries.	Matching Logs with Program Call Paths	Successful	• The initial analysis recognizes 12 potential program call paths that could lead to the recorded program execution. • The final analysis determines that 3 program call paths match the log sequence. • The result is correct, but not exact. • All of the potential program call paths are correct according to system design. • The exactly matched program call paths cannot be determined due to lack of sufficient log entries.
3	• Analyzing a log sequence indicating a system error similar to Case 1. • The log sequence consists of four log entries.	Matching Logs with Sequential Patterns	Successful	• The initial analysis recognizes that the target log sequence matches the pattern of another log sequence related to a similar system error. • The final analysis determines that only 1 program call path matches the log sequence, based on previous analysis of the matched sequential pattern. • The result is correct and exact. • The only matched program call path reveals methods that were related to the cause of the system error.
4	• Analyzing a set of log sequences related to multiple system errors. • Mining for sequential patterns among the log sequences.	Sequential Pattern Mining	Successful	• The mining algorithm managed to identify all potential sequential patterns from the target set. • The implications of identified sequential patterns can be interpreted following the connections between log sequences and program call paths. • The implications correctly reveal insight into the software system in terms of program executions and provide complementary information on defect analysis.
5	• Analyzing a log sequence indicating another system error. • The log sequence consists of four log entries.	Matching Logs with Program Call Paths	Partially Successful	• The initial analysis recognizes over a hundred potential program call paths that could lead to the indicated system error. • The final analysis determines that 9 program call paths match the log sequence. • The result is correct, but not exact. • The result contents are lengthy (over 1000 lines).

Table 9.1: Summary of Experimental Results

Discussion of Effectiveness

In summary, out of *five* problems, *three* were successful and *two* were partially successful.

The successful cases have demonstrated the effectiveness of two different approaches to analyzing a sequence of logs.

Case 1 The approach of matching error logs with program call paths has been applied to analyze a log sequence indicating a system error. The analysis process managed to correctly recognize potential program call paths that could lead to the indicated system error, and effectively determine the program call path that is most closely matched with the sample log sequence.

Case 2 The same approach as in Case 1 has been applied when analyzing a log sequence recording a normal program execution. We managed to achieve an effective analysis result. Both Case 1 and Case 2 have demonstrated that our implementation is capable of providing effective analysis results when analyzing log sequences representing different program executions.

Case 3 The approach of matching error logs with sequential patterns has been applied. It has shown the efficiency and effectiveness of this approach when the log sequence being analyzed actually matches with the pattern of an existing historical log sequence.

Case 4 The sequential pattern mining technique has been applied to explore potential sequential patterns among a set of log sequences. The mining process returned a list of sequential pattens that have been identified. We managed to collect useful information about the software system through observations of these findings.

In practice, the analysis result, such as the matched program call path, can be used by support engineerings as a starting point of their investigation on reported system error. It can save support engineers both the time and effort needed to manually match the sequence of errors logs with the corresponding methods in source code during the investigation.

The partially successful case has shown that our implementation has a certain deficiency. Our log analysis still managed to recognize potential program call paths that are related to the indicated system error, but cannot determine the program call paths that were closely matched with the sample log sequence. We will discuss several factors that could have caused such a deficiency.

Discussion of Deficiency

The analysis results of most sample cases have successfully demonstrated the effectiveness of log analysis; however the analysis result of one case has shown a certain deficiency in our implementation.

Case 5 There was one sequence of four sample logs indicating another system error. The analysis result has over a thousand lines of output and lists over a hundred eligible program call paths. The log analysis has successfully narrowed these down to nine closely matched program call paths. The result has been verified as correct, but not exact. That means, the analysis cannot determine the most closely matched program call path among the potential program call paths that have been identified.

There are several factors contributing to such a deficiency.

- The complexity of source code structure determines the complexity of program call paths. Our partner's software system has complex architecture and a large number of lines of source code. This implies that there can be multiple program call paths between any two methods. The number will grow exponentially when we take more methods into consideration.

- The log analysis relies heavily on information recorded in each log, but in practice, methods could be executed without generating logs. Some methods are not programmed to generate a log at all, which means there is no logging point in the method; some methods have logging points, but due to the restrictions of conditional statements, none of the logging points is executed and as a result, no log is generated. We have to include such methods into the eligible program call paths; as a result, without sufficient logs to match with program call paths, we end up having a large set of most closely matched program call paths.

- The selection of sample logs is essential to log analysis. It is not necessarily true that the more logs for consideration, the better the analysis result. Each program call path has a start point and an end point. Different program call paths can be executed in serial format or parallel format at runtime. It is difficult to identify the start point and the end point of a program call path by manually interpreting lines of logs. When we select a sample sequence of logs, it is possible that the sequence includes logs from two different program executions; as a result, the log analysis may include program call paths from both executions and the result contents are unnecessarily lengthy.

Even though there exists a deficiency in our implementation, the experimental results have sufficiently demonstrated that our methodology and design of the log analysis project provides a concrete and feasible solution to the practical problem of diagnosis of software defects.

9.4 Summary

In this chapter, we have evaluated the implementation of the log analysis project by deploying it on our partner's software system, conducting experiments analyzing sample logs from the system and performing verifications of analysis results. We have demonstrated that the log analysis is able to provide effective results in analyzing a sequence of logs by two approaches: matching error logs with program call paths and matching error logs with sequential patterns. We have also demonstrated that the sequential pattern mining approach can provide extended information about the system from the statistical point of view.

In the final chapter of this thesis we will discuss the contribution and limitations of the work presented in this thesis, as well as the potential future work that can be conducted.

Chapter 10

Conclusion and Future Work

In this chapter, we discuss the contribution of the log analysis project presented in this thesis, the limitations with the current implementation and the future work that may be done to improve it.

10.1 Contribution

We designed and implemented a practical and effective automated diagnostic technique, which combines source code analysis, log analysis and sequential pattern mining, to detect anomalies among logs from a failed program execution, to diagnose reported system errors and to narrow down the range of source code lines to help determine the root cause. We see our main contribution as adopting certain existing work and applying our own methodology and design to integrate all the different implementations and provide a concrete and feasible solution to the practical problems, i.e., diagnosis of software defects.

We implemented the source code analysis to collect information about the software system, including the structure of source code and the logging points in source code that generate logs during program execution. We follow a previously developed process of parsing and analyzing Java bytecode (*class*) files to collect the dependency information among classes and methods and extract method invocations that potentially output logs at runtime. The process involves running an existing off-the-shelf tool to convert Java bytecode into XML representation so that we can parse contents of XML to extract the required information. The process is able to build up the static dependency relations despite working at a relatively high level of abstraction, which are sufficient to provide helpful reference to understanding and analyzing runtime logs during the log analysis process.

We implemented the log analysis to extract information recorded in logs and analyze the information to detect potential system errors and reveal accurate program execution at runtime. An important contribution of our implementation is that we have been able to connect each parsed log with the corresponding logging point in source code. Besides that, we applied graph theory and traversal algorithms to build up the dependency graph following the dependency relations collected in source code analysis. The dependency graph visualizes the structure of the source code and makes it convenient to extract program call paths as the subgraph of the dependency graph. The log analysis process also involves searching for sequential patterns among historical logs, which not only represent the particular execution sequence of methods in the program, but also reflect the potential interrelations between the runtime events recorded by the logs.

The main contribution of the log analysis project is that we have been able to provide two different approaches to analyzing a sequence of logs: matching the sequence of logs with program call paths and matching the sequence of logs with existing sequential patterns. The first approach attempts to match a sequence of logs with the eligible program call paths, in search of the specific program call path that represents the sequence of program executions resulting in the logs. The second approach attempts to match a sequence of logs with existing sequential patterns collected from historical logs, in search of the sequential pattern that represents the sequence of logs and reflects the interrelations between program executions recorded by the logs.

We have integrated an implementation of sequential pattern mining technique into our project, so that we have been able to explore potential sequential patterns among historical logs, reveal interrelations between events recorded by logs and further study the corresponding sequences of program executions behind these events. The sequential pattern mining has been a complementary approach to log analysis. This is another contribution of our project.

The processes discussed above are fully automated. Interactions between the Java classes do not require any human intervention. Moreover, due to the modular design of the processes, if needed, we have the flexibility to stop some of the sub-processes to observe and verify any intermediate outputs. For example, we can print out the dependency relations, the eligible program call paths and the matched program call path during their corresponding analysis processes. The output can be made to a console or a file on disk to verify the correctness of the information.

10.2 Limitations

We discuss several limitations of the log analysis project that we have recognized during the implementation. One limitation of source code analysis is that it currently can handle Java bytecode only, which means software systems developed in languages other than Java are temporarily beyond the scope of our project.

There are a couple of limitations of the dependency graph in source code analysis. One limitation is that at this moment the dependency graph is capable of handling only Java entities inside the Java environment. There are entities that are outside the Java environment but are still accessed during program executions. The dependency relations among such entities are currently not included in the dependency graph. For example, in practice, there can be database accesses from inside the Java methods such as executing database queries. Those database functions and procedures can have dependencies between themselves as well. So the complete dependency graph should extend from the Java environment to the database, which we have left out of the scope of our project. Similarly, there can be web server accesses from inside the Java methods for web applications. The complete dependency graph should include the dependency relations between these external entities as well, which is not presently the case with our project. Another limitation of the dependency graph is that the dependency analysis is limited to the method level, which means the entities in consideration only include classes and methods. This approach makes the analysis process a bit simpler but leaves out much of the crucial data flow and control flow information at the statement level. For example, the dependency graph simply includes the accesses to both branches of an if-else conditional statement as possible program call paths, but is not able to reflect the existence of the statement and the corresponding conditions leading to either of the two branches.

One limitation of log analysis is that the analysis can handle only logs generated by Apache log4j framework. Since most application log formats do not follow a particular standard format, the approach to parsing logs is highly dependent on the semantic structure of the logs. Ideally, knowledge is needed about the expected log format so that it can adopt the corresponding approach to parsing logs. Currently, this has not been implemented in our project. Another limitation of log analysis is the assumption we have made for the logging points. We have to assume that all logging points in the same method should have different constant strings in their log messages, to make it more convenient to match parsed logs with their corresponding logging points. The assumption makes sense in practice, but from the software engineering principle point of view, the implementation should have handled such an issue in a better way.

The sequential pattern mining provides only limited mining results and lacks any practical analysis. It requires manual interpretation and human expertise to obtain insight into the problem behind the sequential patterns identified by the mining algorithm.

10.3 Future Work

Most of the future projects are complementary to the limitations previously mentioned. Regarding the limitation of source code analysis, we should explore feasible options for expanding the source code analysis to include other programming languages in consideration, so that the log analysis can be performed on software systems based on languages other than Java. In future, we may need to handle legacy systems based on older programming languages like COBOL. Iqbal [Iqb11] has suggested that the fact that Java bytecode is a relatively higher level representation than other kinds of compiled code, such as assembly language and machine code, and that several off-the-shelf bytecode analysis tools already exist has facilitated our work to a great extent. In the case of languages like COBOL, we may have to write our own parser and analyzer for processing the compiled representations of these languages, which will be a challenging issue in the future.

As we are concerned about the limitation of the dependency graph to the method level only, its potential extension to the statement level will be a feasible but difficult task. First, it is necessary to understand the Java bytecode representation of both simple and compound statements in Java. Second, we need to find an efficient way to interpret these representations. We can either utilize an existing off-the-shelf bytecode analysis tool that specializes in statement analysis, or we may have to write some kind of parser and analyzer to get the job done. Third, in order to analyze the effect of conditional statements on control flow, besides conventional static analysis, we may need to apply dynamic analysis. Finally, considering all sorts of conditions and restrictions at the statement level, we analyze the control flow and data flow of the program and incorporate any findings in the dependency analysis, so that we can extend the dependency graph to the statement level.

The limitation of log analysis is that it lacks a generic approach to parsing and analyzing logs in a random format. This can be improved in such a way that the implementation reads in a *guidance* on the expected log format before parsing the logs. The guidance basically describes the log semantics and instructs the program to extract information from the corresponding fields of each parsed log; however, this is only for parsing. For analyzing, we not only need the guidance on the log format but also need to know the logging structure in source code, so that we can connect the logs with the corresponding logging points in source code. It will be another challenging future project to make the analysis process a generic approach.

Finally, we consider applying machine learning technique, a branch of artificial intelligence, to further analyze the sequential patterns identified by the mining effort. Specifically, the machine learning technique allows computers to evolve behaviours based on empirical data. A major focus of machine learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on data. This will be a complex but significant experimental project in the future.

Bibliography

[Bal99] T. Ball. The Concept of Dynamic Analysis. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering - ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science (LNCS)*, pages 216–234. Springer Berlin / Heidelberg, 1999.

[Boh02] S.A. Bohner. Extending Software Change Impact Analysis into COTS Components. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 175–182, December 2002.

[CCHK90] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. Constructing the Procedure Call Multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, April 1990.

[CCRS09] T.H. Cormen, C.E.Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, U.S.A., 3rd edition, 2009.

[Cho05] P.K. Chowdhury. Symbolic Interpretation of Legacy Assembly Language. Master's thesis, Department of Computing and Software, McMaster University, August 2005.

[CLM+09] T.M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, and K. Vaswani. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Proceedings of 31st IEEE International Conference on Software Engineering 2009 (ICSE '09)*, pages 34–44, Vancouver, Canada, May 2009.

[Cor12] Oracle Corporation. MySQL :: The world's most popular open source database, February 2012. Electorinally available at http://www.mysql.com.

[Eck95] W.W. Eckerson. Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. *Open Information Systems*, 3(20):46–50, January 1995.

[Ern03] M.D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In J. Cook and M. Ernst, editors, *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, Oregon, May 2003. ICSE'03.

[Fai78] R.E. Fairley. Tutorial: Static Analysis and Dynamic Testing of Computer Software. *Computer*, 11(4):14–23, April 1978.

[Fou12a] The Apache Software Foundation. Apache Ant, February 2012. Electronically available at http://ant.apache.org/.

[Fou12b] The Apache Software Foundation. Apache log4j 1.2 - Short Introduction to log4j, February 2012. Electronically available at http://logging.apache.org/log4j/1.2/manual.html.

[Fou12c] The Apache Software Foundation. Apache Logging Service Project

- Apache log4j, February 2012. Electronically available at http://logging.apache.org/log4j/.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.

[FP98] N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomsen Publishing Inc., 2nd edition, 1998.

[FV12] P. Fournier-Viger. SPMF: A Sequential Pattern Mining Framework, February 2012. Electorinally available at http://www.philippe-fournier-viger.com/spmf/.

[Gra85] J. Gray. Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7, June 1985.

[HPMA+00] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M-C. Hsu. FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining. In *Proceedings of 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 355–359, New York, NY, USA, 2000. ACM.

[HPY05] J. Han, J. Pei, and X. Yan. Sequential Pattern Mining by Pattern-Growth: Principles and Extensions. In W. Chu and T.Y. Lin, editors, *Foundations and Advances in Data Mining*, volume 180 of *Studies in*

Fuzziness and Soft Computing, pages 183–220. Springer Berlin / Heidelberg, 2005.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing using Dependence Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, January 1990.

[Iqb11] A. Iqbal. Identifying Modifications and Generating Dependency Graphs for Impact Analysis in a Legacy Environment. Master's thesis, Department of Computing and Software, McMaster University, April 2011.

[Jan06] B.J. Jansen. Search log analysis: What it is, what's been done, how to do it. *Library & Information Science Research*, 28:407–432, 2006.

[Kas04] H. Kastenberg. Software Metrics as Class Graph Properties. Master's thesis, Department of Electrical Engineering, Mathematics and Computer Science, University of Twente, July 2004.

[LAZJ03] B. Liblit, A. Aiken, A.X. Zheng, and M.I. Jordan. Bug Isolation via Remote Program Sampling. *ACM SIGPLAN Notices*, 38(5):141–154, May 2003.

[LH96] L. Larsen and M.J. Harrold. Slicing Object-Oriented Software. In *Proceedings of the 18th International Conference on Software Engineering 1996*, pages 495–505, Berlin, Germany, March 1996.

[LIT91] I. Lee, R.K. Iyer, and D. Tang. Error/Failure Analysis Using Event Logs from Fault Tolerant Systems. In *Proceedings of Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., 21st International Symposium*, pages 10–17, Montreal, Quebec, Canada, June 1991.

[LY12] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, February 2012. Electorinally available at http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html.

[MCD09] T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred Call Path Profiling. *ACM SIGPLAN Notices*, 44(10):175–190, October 2009.

[MD12] J. Meyer and T. Downing. Java Virtual Machine - Online Instruction Reference, February 2012. Electorinally available at http://cs.au.dk/ mis/dOvs/jvmspec/ref-Java.html.

[MMKM94] B.A. Malloy, J.D. McGregor, A. Krishnaswamy, and M. Medikonda. An Extensible Program Representation for Object-Oriented Software. *ACM SIGPLAN Notices*, 29(12):38–47, December 1994.

[OO84] K.J. Ottenstein and L.M. Ottenstein. The Program Dependence Graph in a Software Development Environment. *ACM SIGSOFT Software Engineering Notes*, 9(3):177–184, April 1984.

[PHMA$^+$01] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In *Proceedings of International Conference on Data Engineering 2001 (ICDE‘01)*, pages 215–224, Heidelberg, Germany, April 2001.

[PHMA+04] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1424–1440, November 2004.

[RB83] R.E. Rice and C.L. Borgman. The Use of Computer-Monitored Data in Information Science and Communication Research. *Journal of the American Society for Information Science and Technology (JASIST)*, 34(4):247–256, July 1983.

[Ree00] G. Reese. *Database Programming with JDBC and Java.* O'Reilly and Associates, 2nd edition, November 2000.

[Ryd79] B.G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[SA96] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In P. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology - EDBT '96*, volume 1057 of *Lecture Notes in Computer Science (LNCS)*, pages 1–17. Springer Berlin / Heidelberg, 1996.

[Tes12a] J. Tessier. Dependency Finder, February 2012. Electronically available at http://depfind.sourceforge.net/.

[Tes12b] J. Tessier. The Dependency Finder User Manual, February 2012. Electorinally available at http://depfind.sourceforge.net/Manual.html.

[XHF+09a] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Online System Problem Detection by Mining Patterns of Console Logs. In *Proceedings of 9th IEEE International Conference on Data Mining 2009 (ICDM '09)*, pages 588–597, Miami, FL, USA, December 2009.

[XHF+09b] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. Jordan. Large-Scale System Problems Detection by Mining Console Logs. Technical Report UCB/EECS-2009-103, July 2009. Electronically available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-103.html.

[YHA03] X. Yan, J. Han, and R. Afshar. CloSpan: Mining Closed Sequential Patterns in Large Datasets. In *Proceedings of 3rd SIAM International Conference on Data Mining*, pages 166–177, May 2003.

[YMX+10] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. ASPLOS'10, March 2010.

[Zak01] M.J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning*, 42(1/2):31–60, Jan/Feb 2001.

[Zha98] J. Zhao. Applying Program Dependence Analysis To Java Software. In *Proceedings of Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, pages 162–169, 1998.