UNIVERSITY OF CALGARY

Characterization of Logging Usage:
An Application of Discovering Infrequent Patterns via antiunification

by

Narges Zirakchianzadeh

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

August, 2016

© Narges Zirakchianzadeh 2016

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Characterization of Logging Usage: An Application of Discovering Infrequent Patterns via antiunification" submitted by Narges Zirakchianzadeh in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Dr. Robert J. Walker
Supervisor
Department of Computer Science

Dr. Jörg Denzinger
Examiner
Department of Computer Science

Dr. Christian J Jacob
Examiner
Department of Computer Science

Date

Abstract

Acknowledgements

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Abbreviation	Definition
AST	Abstract Syntax Tree
AU	Anti-unification
AUAST	Anti-unification Abstract Syntax Tree
HOAUMT	Higher-order Anti-unification Modulo Theories
LJC	Logged Java Class

Chapter 1

Introduction

Understanding the similarities and differences of a set of source code fragments is a potentially complex problem that has various actual or potential applications in program analysis such as collating application programming interface (API) usage patterns, detecting code clones [Bulychev and Minea, 2009], automating source code reuse [Cottrell et al., 2008], recommending replacements for APIs between various versions of a software library [Cossette et al., 2014], and automating the merge operation of various branches in a version control system. As a specific application, the focus of this study is on characterizing where logging is used in the source code by understanding the structural correspondences and differences of a set of source code fragments enclosing logging calls within a software system or from different software systems.

Logging is a conventional programming practice to record an application's state and/or actions during the program's execution [Gupta, 2005], and log system analysis assist developers in diagnosing the presence or absence of a particular event, understanding the state of an application, and following a programs execution flow. The importance of logging has been identified by its various applications in software development and maintenance tasks such as problem diagnosis [Lou et al., 2010], system behavioral understanding [Fu et al., 2013], quick debugging [Gupta, 2005], performance diagnosis [Nagaraj et al., 2012], easy software maintenance [Gupta, 2005], and troubleshooting [Fu et al., 2009].

Developers can perform logging in various ways as they could make different decisions about where and what to log. For example, they can apply logging to record the ocuurance of every event of an application and use logging calls at the start and end of the body of every method in the source code [Clarke et al., 1999a,b]. However, three main problems are associated with excessive logging. First, it can generate a lot of redundant information that might be confusing and mis-

leading for developers to perform system log analysis, masking significant information. Second, excessive logging is costly. It requires extra time and effort to write, debug, and maintain the logging code. Third, it can cause system resource overhead and thus the application performance will be negatively affected. On the other hand, insufficient logging may result in losing some necessary run-time information for software analysis. Therefore, logging should be done in an appropriate manner to be effective.

Despite the importance of logging for software development and maintenance, few studies have been conducted on understanding logging usage in real-world applications since logging has been considered as a trivial task [Clarke et al., 1999a,b]. However, the availability of several complex frameworks (e.g., `Log4J`,`SLF4J`) that assists developers to log suggests that effective logging is not a straightforward task to perform in practice. In addition, Yuan et al. [2012a] study shows that developers spend a great effort to modify logging practices as after-thoughts, which indicates that it is not that simple for developers to perform logging practices efficiently in their first attempt.

In this research, we would like to understand where developers log in practice in a detailed way. The location of logging calls has a great impact on the quality of logging as it helps developers to trace the code execution path to identify the root causes of an error in log system analysis. However, to the best of our knowledge, no previous work has focused on characterizing where logging calls are used in real-world applications. In this study, we addresses this gap by developing an automated approach to detect the detailed structural correspondences and differences in the usage of logging within a system and between systems.

1.1 Programmatic support for logging

A typical logging call is composed of a log function and its parameters including a log text message and verbosity level. A log text message consists of static text to describe the logged event and some optional variables related to the event. The verbosity level is intended to classify the severity of a logged event such as a debugging note, a minor issue, or a fatal error. Figure 1.1 provides

examples of logging calls from the `log4j` framework in descending order of severity. The `fatal` level designates a very severe error event that will likely lead the application to terminate. The `error` level indicates that a non-fatal but clearly erroneous situation has occurred. The `warn` level indicates that the application has encountered a potentially harmful situation. The `info` level designates important information that might be helpful in detecting root cases of an error or to understand the application behaviour. The `debug` level designates useful information to debug an application and is usually used by developers only during the development phase. In general, verbosity level is used for classification to avoid the overhead of creating large log files in high performance code.

```
log. fatal (”Fatal  Message %s”, variable);
log. error (”Error  Message %s”, variable);
log.warn(”Warn Message %s”, variable);
log. info (”Info  Message %s”, variable);
log.debug(”Debug Message %s”, variable);
```

Figure 1.1: Logging call examples from the `Log4j` framework

1.2 Overview of related work

Research on the problem of characterizing logging practices can be divided into two main topics: context and location of logging calls. The context refers to the log text messages and the location refers to where logging calls are used in the source code. Few studies have been conducted on characterizing log message modifications [Yuan et al., 2012a] and developing tools to automatically enhance existing log messages [Yuan et al., 2012b, 2010]. However, to the best of our knowledge, no research has been conducted on studying the location of logging calls in real-world software systems.

Understanding the commonalities and differences between source code fragments has been

used for various applications (e.g., [Cottrell et al., 2007, 2008, 2009, Cossette et al., 2014, Bulychev and Minea, 2009]). However, our study makes the first attempt to characterize the usage of logging calls by automatically detecting the detailed structural correspondences and differences of a set of source code fragments enclosing them.

1.3 Broad thesis overview

In this study, we aim to provide a concise description of where logging is used in the source code by constructing generalizations that represents the detailed structural similarities and differences between entities that make use logging calls. Our study investigates the location of logging calls from the point of view of logged Java methods (LJM) which are the Java methods enclosing logging calls in source code. To investigate how to construct generalizations using the syntax and semantics of the Java programming language, we looked to a previous research conducted by Cottrell et al. [2008] that determines the detailed structural correspondences between two Java source code fragments through the application of approximated anti-unification such that one fragment can be integrated to the other one for small scale code reuse. However, our problem context is different as we need to generalize a set of source code fragments with special attention to logging calls. Therefore, our approach must take into account the logging calls when performing the generalization task via the determination of structural correspondences.

Our approach employs a hierarchical clustering algorithm to create a generalization hierarchy from a set of LJMs using a measure of similarity. It uses an approximated anti-unification algorithm to construct a structural generalization representing the similarities and differences between a pair of LJMs. Our anti-unification approach proceeds in three steps. First, it uses the Jigsaw framework [Cottrell et al., 2008] to determine all potential correspondences between the two LJMs using a measure of similarity that relies on structural correspondences along with a simple knowledge of semantic equivalences in the Java language specifications. Second, it develops a greedy selection algorithm to approximate the best anti-unifier to our problem by determining the most

similar correspondence for each substructure in our structures. It also applies some constraints in determining correspondences to prevent the anti-unification of logging calls with anything else. Third, it constructs an anti-unifier through the anti-unification of two structures and develops a measure of structural similarity between them.

1.4 Thesis Statement

The thesis of this work is to determine the detailed structural similarities and differences between entities of the source code that make use of logging calls to provide a concise description of where logging do occur in real-world software systems.

1.5 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 motivates the problem of understanding where to use logging calls in the source code through a scenario in which a developer attempts to perform a logging task. This scenario outlines the potential problems she may encounter and illustrates that the current logging practice is insufficiently supported.

Chapter 3 provides background information on abstract syntax trees (ASTs), which are the basic structure we will use for describing software source code. We also provide a definition of anti-unification and why it does not adequately address our problem. Then we define higher-order anti-unification modulo theories (HOAUMT), an extension to anti-unification that can be applied on an extended form of AST to solve our problem. Afterwards we discuss the Jigsaw framework, an existing tool that could assist us in the determination of potential structural correspondences by applying HOAUMT on extended structures.

Chapter 4 describes our proposed approach and its implementation as a plug-in to the Eclipse integrated development environment (IDE). Chapter 5 presents two empirical studies. The first study is conducted to evaluate the accuracy of our approach and its implemented tool by conducting an experiment on 10 sample logged Java methods. The second study is conducted to characterize

the location of logging usage in three open-source software systems.

Chapter 6 discusses the results and findings of my work, threats to its validity, and the remaining issues. Chapter 7 describes related work to our research problem and how it does not adequately address the problem. Chapter 8 concludes the dissertation and presents the contributions of this study and future work. Additional materials appended to the end of this dissertation is provided in Appendix A.

Chapter 2

Motivational Scenario

Printing messages to the console or to a log file is an integral part of a software development that can be used to test, debug, and understand whats going on inside an application. In Java programming language, `print()` statements are commonly used to print something on console. However, the availability of tools, frameworks and APIs for logging that offers more powerful and advanced Java logging features, flexibility, and improvement on logging quality suggests that using the `print()` statements is not sufficient for real-world applications.

The logging framework offers a lot more features, which is not possible using the `print()` statements. In most logging frameworks (e.g., `Log4j` or `SLF4j`, `java.util.logging`) , different verbosity levels of logging are available for use. That is, by logging at a particular log level, messages will get logged for that level and all levels above it and not for the levels below. As an example, `debug` log level messages can be used in test environment, while `error` log level messages can be used in production environment. This feature not only produces fewer log messages but also improves the performance of an application. In addition, most logging frameworks allow the production of formatted log messages, which makes it easier to monitor the behaviour of a system by a developer. Furthermore, in case of working on a server side application then only way to know what is going on inside the server is by monitoring log file. Although logging is a precious practice for software development and maintenance, it imposes extra time and energy on developers to write, test, and run the code, while affecting the application performance. Since latency and speed is major concerns for most software systems, it becomes necessary to understand and learn logging in great details in order to perform logging in an efficient manner.

To illustrate the challenges that lie in effectively performing logging practices in software systems, consider a scenario in which a developer is asked to log an event based mechanism of a text

editor tool written in Java programming language. Consider the developer trying to log a Java class of this system shown in Figure 2.1 using the `log4j` logging framework. She knows that components of this application register with the `EditBus` class to receive messages reflecting changes in the application's state, and the `EditBus` class maintains a list of components that have requested to receive messages. That is, when a message is sent using this class, all registered components receive it in turn. Furthermore, any classes that subscribes to the `EditBus` and implements the `EBComponent` interface defines the method `EBComponent.handleMessage(EBMessage)` to handle a message sent on the `EditBus`. To perform this logging task several fundamental questions might appears in her mind which are mostly related to where and what to log.

```
1 public class EditBus {
2   private static ArrayList components=new ArrayList();
3   private static EBComponent[] copyComponents;
4   private EditBus(){
5   }
6   public static void addToBus(EBComponent comp){
7     synchronized (components) {
8       components.add(comp);
9       copyComponents=null;
10    }
11  }
12  public static void removeFromBus(EBComponent comp){
13    synchronized (components) {
14      components.remove(comp);
15      copyComponents=null;
16    }
17  }
18  public static EBComponent[] getComponents(){
19    synchronized (components) {
20      if (copyComponents == null) {
21        copyComponents=(EBComponent[])components.toArray(new EBComponent[
              components.size()]);
22      }
23      return copyComponents;
24    }
25  }
26  public static void send(EBMessage message){
27    EBComponent[] comps=getComponents();
28    for (int i=0; i < comps.length; i++) {
29        EBComponent comp=comps[i];
30        long start=System.currentTimeMillis();
31        comp.handleMessage(message);
32        long time=(System.currentTimeMillis() - start);
33    }
34  }
35 }
```

Figure 2.1: The `org.gjt.sp.jedit.EditBus` class

The first solution she can come up with is to simply log at the start and end of every method. However, she believes that logging at the start and end of the `addToBus(EBComponent)`, `removeFromBus(EBComponent)`, and `getComponents()` methods are useless, producing redundant information. She assumes that the more she logs, the more she performs file IO

which slows down the application. Therefore, she decides to log only important information which is necessary to debug or troubleshoot potential problems if they happen. She proceeds to identify the information needed to be logged and then decides on where to use logging calls. She thinks that it is important to log the information related to a message sent to a registered component, including the message content and the transmission time, to troubleshoot the potential problems that might happen in sending messages. She simply wants to begin with using a logging call at the start of the `send` method (Line 2 of Figure 2.2) to log the information. However, she realizes that this logging call does not allow her to log the information she wants as the `time` variable is not initialized in the beginning of this method, thus she proceeds to examine the body of the `send` method line-by-line and uses another logging call after the `time` variable is initialized inside an **if** statement that checks the value of the variable `time` is not invalid (shown in Lines 9-11 of Figure 2.3).

```
1 public static void send(EBMessage message){
2     //logging call
3     EBComponent[] comps=getComponents();
4     for (int i=0; i < comps.length; i++) {
5         EBComponent comp=comps[i];
6         long start=System.currentTimeMillis();
7         comp.handleMessage(message);
8         long time=(System.currentTimeMillis() − start);
9     }
10 }
```

Figure 2.2: The developers determination of the usage of logging calls for the `EditBus.send(EBMessage)` method.

```
1 public static void send(EBMessage message){
2     //logging call
3     EBComponent[] comps=getComponents();
4     for (int i=0; i < comps.length; i++) {
5         EBComponent comp=comps[i];
6         long start=System.currentTimeMillis();
7         comp.handleMessage(message);
8         long time=(System.currentTimeMillis() − start);
9         if (time != 0){
10            //logging call
11        }
12    }
13 }
```

Figure 2.3: The developers determination of the usage of logging calls for the `EditBus.send(EBMessage)` method.

She also believes that it is important to log an error if any problems happen in sending messages to the components. She decides to use a **try/catch** statement as it is a common way to handle exceptions in the Java programming language. She creates a **try/catch** block to capture the potential failure in sending messages and uses a logging call inside the **catch** block to log the exception (shown in Lines 2-16 of Figure 2.4). However, she realizes that using this logging call would not allow her to reach to the desired functionality as it does not reveal that the problem is related to which component. Thus, she decides to re-locate the **try/catch** block inside the **for** statement to log an error in case of a problem in sending messages to any components (shown in Lines 5-15 of Figure 2.5).

```
1  public static void send(EBMessage message){
2      try{
3          //logging call
4          EBComponent[] comps=getComponents();
5          for (int i=0; i < comps.length; i++) {
6              EBComponent comp=comps[i];
7              long start=System.currentTimeMillis();
8              comp.handleMessage(message);
9              long time=(System.currentTimeMillis() − start);
10             if (time != 0){
11                 //logging call
12             }
13         }
14     }catch (Throwable t) {
15         // logging call
16     }
17 }
```

Figure 2.4: The developers determination of the usage of logging calls for the `EditBus.send(EBMessage)` method.

```
1  public static void send(EBMessage message){
2      //logging call
3      EBComponent[] comps=getComponents();
4      for (int i=0; i < comps.length; i++) {
5          try{
6              EBComponent comp=comps[i];
7              long start=System.currentTimeMillis();
8              comp.handleMessage(message);
9              long time=(System.currentTimeMillis() − start);
10             if (time != 0) {
11                 //logging call
12             }
13         }catch (Throwable t) {
14             // logging call
15         }
16     }
17 }
```

Figure 2.5: The developers determination of the usage of logging calls for the `EditBus.send(EBMessage)` method.

```
1 public class EditBus {
2  private static  ArrayList  components=new ArrayList();
3   private static  EBComponent[] copyComponents;
4   private EditBus(){
5   }
6   public static  void addToBus(EBComponent comp){
7     synchronized (components) {
8       components.add(comp);
9       copyComponents=null;
10    }
11  }
12  public static  void removeFromBus(EBComponent comp){
13    synchronized (components) {
14      components.remove(comp);
15      copyComponents=null;
16    }
17  }
18  public static  EBComponent[] getComponents(){
19    synchronized (components) {
20      if  (copyComponents == null) {
21        copyComponents=(EBComponent[])components.toArray(new EBComponent[
              components.size()]);
22      }
23      return copyComponents;
24    }
25  }
26  public static  void send(EBMessage message){
27    //logging  call
28    EBComponent[] comps=getComponents();
29    for  (int  i=0;  i  <  comps.length; i++) {
30      try{
31        EBComponent comp=comps[i];
32        long start=System.currentTimeMillis();
33        comp.handleMessage(message);
34        long time=(System.currentTimeMillis() − start);
35        if  (time != 0)  {
36          //logging  call
37        }
38      }catch (Throwable t) {
39        //  logging  call
40      }
41   }
42  }
43 }
```

Figure 2.6: The developers determination of the usage of logging calls for the `org.gjt.sp.jedit.EditBus` class.

Figure 2.6 shows the developers determination of the usage of logging calls to perform logging task of the `org.gjt.sp.jedit.EditBus` class. With taking proper decisions about where to use logging calls, the developer is in good position to proceed to write the logging messages by examining the remaining conceptually complex questions. Which information should I log? How to choose the format of log message? Which information goes to which level of logging? If she had reached this point more easily and quickly, she would have had more time and energy to make decisions about the remaining issues to complete the logging practice in a timely and appropriate manner.

To sum up, this scenario involves the developer to make a large collection of decisions and then act on them. Her attention is split between both the detailed and the high-level decision. In addition, time constraints and performing tedious tasks can cause the developer to make bad decisions. However, having a vision of where usually developers use logging calls in similar situations could guide her to make informed decision about where to use logging calls and thus perform the logging task in a faster and less error-prone manner.

2.1 Summary

This motivational scenario highlights the problems a developer may encounter to perform a logging task. The core problem she faces in this scenario is the difficulty in understanding where to use logging calls that enables her to log the desired information. However, having an understanding of how usually developers log in similar situations might assist her to make informed decisions about where to use logging calls more quickly, and so she could pay more attention to the remaining conceptually complex issues to complete the logging task.

Chapter 3

Background

The structure of a program can be described using its syntax and a Java source code can be represented as an instance of an Abstract Syntax Tree. **[RW: No. Any program can be represented as an abstract syntax tree.]** To construct structural generalizations describing the correspondences and differences between logged Java classes, first we should understand what AST is and how specific information about each Java element is held in AST structure. Then we should investigate the application of antiunification and its extensions on this structure to produce structural generalizations. We should also figure out how the Jigsaw framework could assist us in determining potential candidate structural correspondences.

Antiunification is summarized in section 3.1 starting with an introduction to unification and its dual antiunification and followed by a discourse regarding to limitations of antiunification to address our problem. Sections 3.2 and 3.3 of this chapter establish a brief description of the Abstract Syntax Tree (AST) structure and its extended form, necessary to understand the requirements that guided the development of an antiunification algorithm for our application. Section 3.4 is dedicated to explain higher-order antiunification modulo theories, an extension to antiunification, in which a set of equivalence theories are defined and applied on higher-order extended structures to incorporate background knowledge. Afterwards we discuss the Jigsaw framework and its application in determining potential candidate structural correspondences in Section 3.5.

3.1 Antiunification

[RW: This section needs to come after the section on ASTs.]

To describe unification theory and its dual antiunification theory, we first introduce a formal definition of terms, the application of a substitution on a term, and the definition of instance and

anti-instance of a term, as the requirements needed to understand the theories.

Definition 3.1.1 (Term)**.** A term is a set of function symbols, variables, and constants, such that function symbols can come up with unlimited number of arguments. **[RW: "Set" implies no order and that these items cannot repeat. This is wrong. "Come up with" doesn't make much sense here.]**

We represent function symbols by identifiers starting with a lowercase letter (e.g., $f(a, b)$), variables are represented by identifiers starting with an uppercase letter (e.g., X, Y), and constants are function symbols with no arguments (e.g., a, b). the followings are examples of term:

- Y
- a
- $f(X, c)$
- $f(g(X, b), Y, g(a, Z))$

Definition 3.1.2 (Applying substitutions)**.** A substitution is a mapping from variables to terms, and the application of a substitution to a term would result in replacing all occurrences of each variable in the term by a proper subterm.

As an example, an application of a substitution $\Theta = X \rightarrow a$ on a term $f(X, b)$ is defined by replacing all occurrences of the variable X by the term a and thus $f(X, b) \xrightarrow{\Theta} f(a, b)$.

Definition 3.1.3 (instance & anti-instance)**.** a is an instance of a term X and X is an anti-instance of a, if there is a substitution Θ such that the application of Θ on X results in a (i.e., $X \xrightarrow{\Theta} a$).

Definition 3.1.4 (Unifier)**.** An unifier is a common instance of two given terms.

Unification usually aims to create the *most general unifier* (MGU); that is, U is the MGU of two terms such that for all unifiers U' there exists a substitution Θ such that $U \xrightarrow{\Theta} U'$. Unification has been used for various applications; however, it is not helpful to solve our problem as we need

to construct generalizations based on the following description: **[RW: The following description does not justify the claim that unification is not helpful.]**

Definition 3.1.5 (Generalization)**.** X is a generalization for a and b, where X is an anti-instance for a and b under substitutions Θ_1 and Θ_2, respectively (i.e., $X \xrightarrow{\Theta_1} a$ and$X \xrightarrow{\Theta_2} b$.

To create a generalization of two given terms, we should use the inverse of unification, which is called *antiunification*, where the two original terms would be instances of the new antiunified term.

Definition 3.1.6 (antiunifier)**.** An antiunifier is a common generalization of two given terms.

An antiunifier contains common pieces of the original terms, while the differences are abstracted away using variables. An antiunifier for a pair of terms always exists since we can antiunify any two terms by creating a variable X. However, antiunification usually aims to find the *most specific antiunifier* (MSA); that is , A is the MSA of two structures where there exists no antiunifier A' such that $A \xrightarrow{\Theta} A'$.

As an example, the antiunifier of two given terms $f(X, b)$ and $f(a, Y)$ is the new term $f(X, Y)$, containing common pieces of the two original terms. The variable Y in the antiunifier $f(X, Y)$ can be substituted by the term b to re-create $f(X, b)$ (with $\Theta_1 = Y \rightarrow b$) and the variable X in the antiunifier can be substituted by the term a to re-create $f(a, Y)$ (with $\Theta_2 = X \xrightarrow{\Theta} a$), as depicted in Figure 3.1. In addition, the unifier $f(a, b)$ of the two terms can be instantiated by applying the substitutions $\Theta'_1 = X \xrightarrow{\Theta} a$ and $\Theta'_2 = Y \xrightarrow{\Theta} b$ on the terms $f(X, b)$ and $f(a, Y)$, respectively.

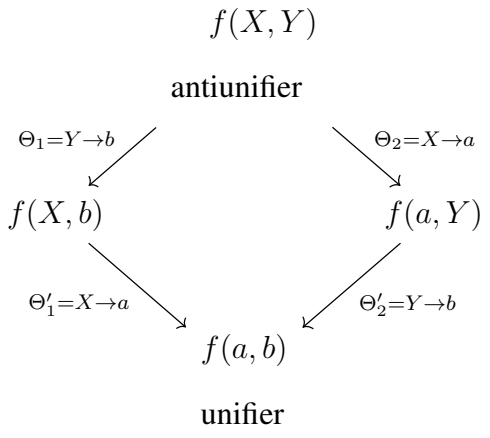

Figure 3.1: The unification and antiunification of the terms $f(X,b)$ and $f(a,Y)$.

MSA should preserve as much of common pieces of both original terms as possible, however, antiunification fails to capture complex commonalities as it restricts substitutions to replace only first-order variables by terms. That is, when two terms differ in function symbols, antiunification fails to capture common details of them. For example, the antiunifier of the terms $f(a,b)$ and $g(a,b)$ is X using antiunification as depicted in Figure 3.2.

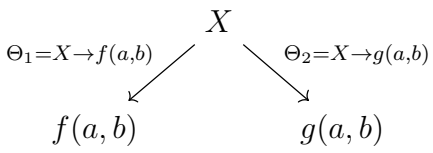

Figure 3.2: The antiunification of the terms $f(X,b)$ and $f(a,Y)$.

An extended form of antiunification, which is called higher-order antiunification, would allow us to create MSA by extending the set of possible substitutions such that variables can be replaced by not only constants but also functional symbols to retain the detailed commonalities. For example, the antiunifier of the terms $f(a,b)$ and $g(a,b)$ is $X(a,b)$ using higher-order antiunification as depicted in Figure 3.3.

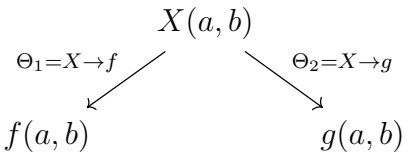

Figure 3.3: The higher-order antiunification of the terms $f(X,b)$ and $f(a,Y)$.

In the following sections, we will provide a brief description of AST structures and the application of antiunification on its extended form to construct structural generalizations.

3.2 Abstract Syntax Tree

The Eclipse Java Development Tools (JDT) framework provides APIs to access and manipulate Java source code via Abstract Syntax Tree (AST). AST maps Java source code in a tree structure form and thus every Java source code can be represented as tree of AST nodes, where each represents an element of the Java Programming Language. AST helps developers to modify and analyze the Java program in a more convenient way than text-bases source code by providing a language parser of the Java source code, determining the bindings between name and type references, and providing specific information of each Java element. For example, the simple AST structure of two sample logged Java classes in Figures 3.4 an 3.5 is shown in Figure 3.6

```
1 public abstract class EBPlugin extends EditPlugin implements EBComponent {
2   private Boolean seenWarning;
3   protected EBPlugin(){
4   }
5   public void handleMessage( EBMessage message){
6     if  (seenWarning)    return;
7     seenWarning=true;
8     Log.log(Log.WARNING,this,getClassName() + ” should extend” + ” EditPlugin not
          EBPlugin since it has an empty”+ ” handleMessage()”);
9   }
10 }
```

Figure 3.4: A Java class that uses a logging call. This will be referred to as Example 1.

```
1 public static class Wrapper implements ActionListener {
2   private ActionContext context;
3   private String actionName;
4   public Wrapper( ActionContext context,   String actionName){
5     this.context=context;
6     this.actionName=actionName;
7   }
8   public void actionPerformed( ActionEvent evt){
9     EditAction action=context.getAction(actionName);
10    if (action == null) {
11      Log.log(Log.ERROR,this,"Unknown action: " + actionName);
12    }
13    else      context.invokeAction(evt,action);
14  }
15 }
```

Figure 3.5: A Java class that uses a logging call. This will be referred to as Example 2.

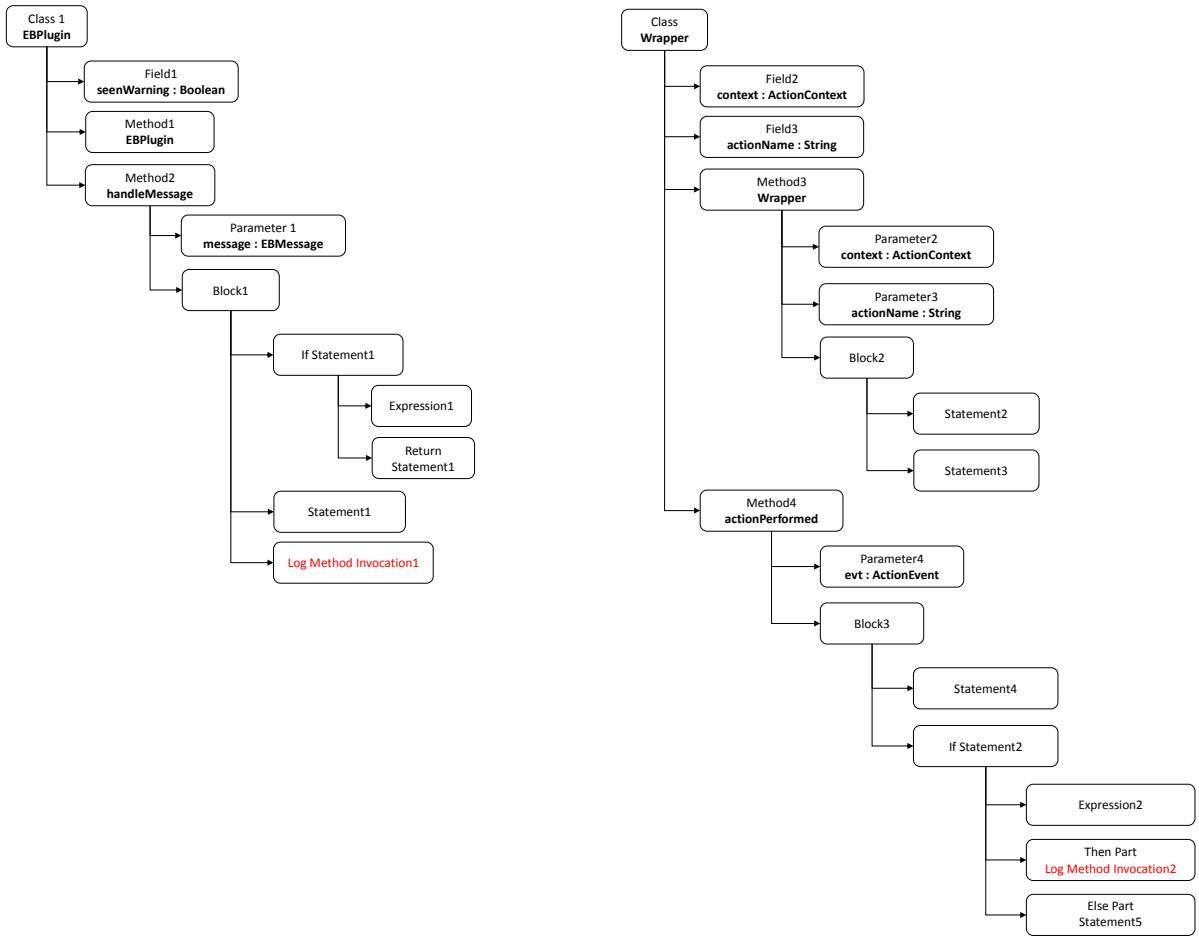

Figure 3.6: Simple AST structure of examples in Figures 3.4 and 3.5.

In the JDT framework, structural properties of each AST node can be used to obtain specific information of the Java element it represents. These properties are stored in a map data structure that associates each property to its value and are divided into three types:

- *Simple structural properties:* that contain a simple value which has a primitive or simple type or a basic AST constant (e.g., identifier property of a name node whose value is a String)
- *Child structural properties:* where the value is a single AST node (e.g., name property of a method declaration node)

- *Child list structural properties*: where the value is a list of child nodes (e.g., body declarations property of a class declaration node whose value is a list of body declaration nodes, including method declaration and field declaration nodes.)

An instance of an AST structure can be represented in an abstract form that can be mapped to the definition of a term described in Section 3.1. As an example, the abstract form of ASTs of logging calls of Java classes in Figures 3.4 and 3.5 can be represented as:

- *expression(expression(Log),name(log),arguments(leftoperand(message),+, rightoperand(" is empty"),qualifier(Log),name(WARNING)))*

- *expression(expression(Log),name(log),arguments(leftoperand(actionName), +,rightoperand("is an unknown action"),qualifier(Log),name(WARNING)))*

Where ASTNodes (e.g., *expression, name,, qualifier*) might be viewed as function symbols and simple values (e.g., *log, WARNING*) might be viewed as constants in the term definition. As described in Section 3.1, antiunification utilizes variables that must be substituted with proper structures to re-create original structures. However, the AST structure does not contain any variables and so we need to construct an extended form of AST, which will be described in the following section.

3.3 Constructing the AUAST

AUAST (antiunified AST) is an extended form of AST that allows the insertion of variables in place of any node in the tree structure, including both subtrees and leaves, to indicate variations between original structures. The AUAST addresses the limitations of AST to construct an antiunifier by adding the following structural properties:

- *Simple Variable Property*: an extension of simple property referring to two simple values to allow the insertion of variables in place of leaves.

- *Child Variable Property*: an extension of child property referring to two child AST nodes to allow the insertion of variables in place of subtrees.

The antiunification of AUASTs of logging calls in Figures 3.4 and 3.5 is depicted in Figure 3.7. The new variables X and Y are created to abstract away the structural variations.

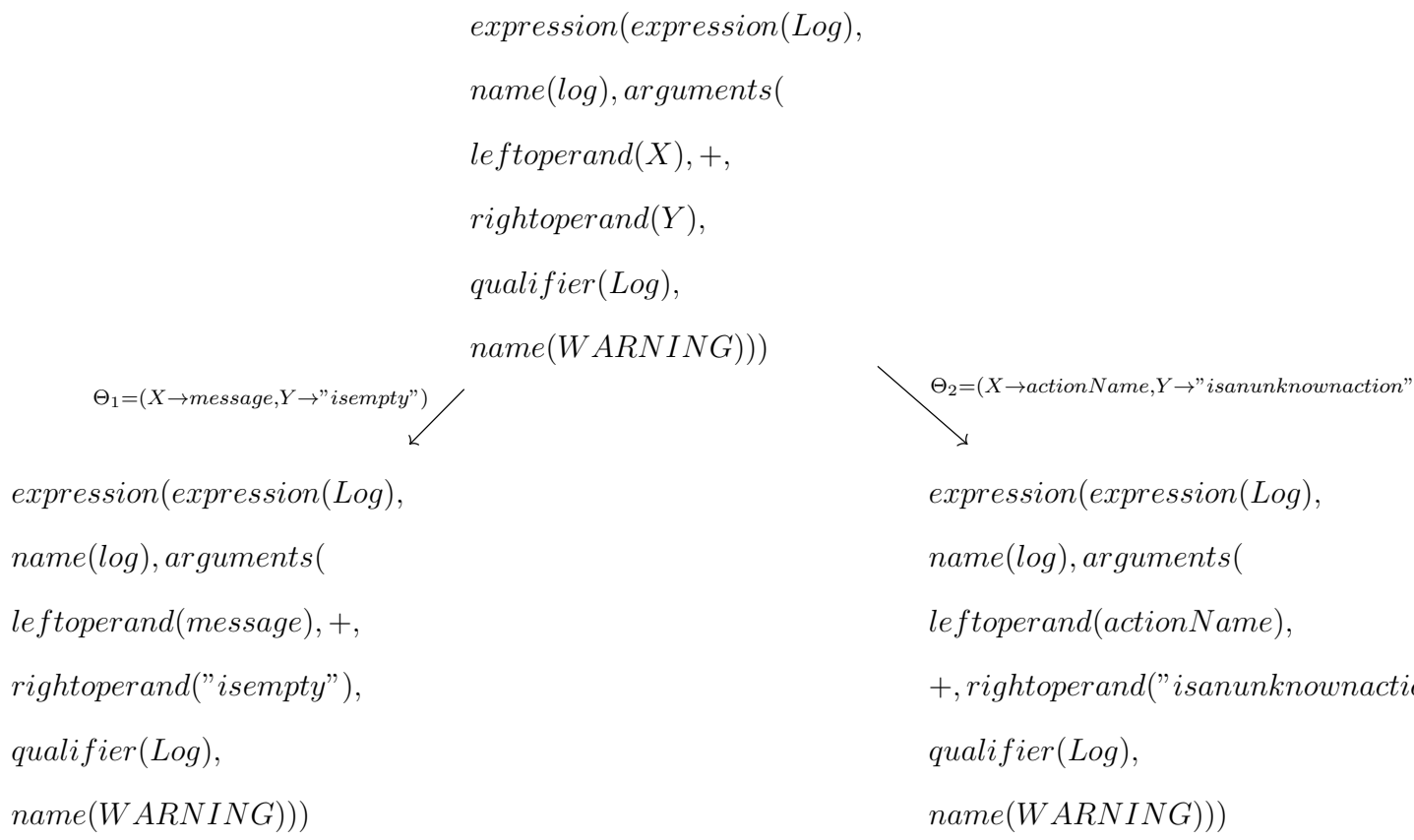

Figure 3.7: The antiunification of AUASTs of logging calls in the Examples 1 and 2.

Applying higher-order antiunification on AUAST structures could help us in constructing a structural generalization by maintaining the common pieces and abstracting the differences away using variables. However, it is not comprehensive enough to solve our problem as it does not consider background knowledge about AST structures, such as syntactically different but semantically relevant structures, missing structures, and different ordering of arguments. In the following sec-

tion, we will look at an extension of antiunification, higher-order antiunification modulo theories, and how it can sufficiently address the limitations of antiunification in our context.

3.4 Higher-order antiunification modulo theories

In higher-order antiunification modulo theories, a set of equivalence equations is defined to incorporate background knowledge. Each equivalence equation $=_E$ determines which terms are considered equal and a set of these equations can be applied on higher-order extended structures to determine structural equivalences. For example, we have introduced an equivalence equation $=_E$, such that $f(X, Y) =_E f(Y, X)$ to indicate that the ordering of arguments does not matter in our context.

We have also introduced a theory, called NIL-theory, that adds the concept of NIL structure, which is defined to create a structure out of nothing, and defines an equivalence equation $=_E$ for it. The NIL structure can be used to antiunify two structures when a substructure exists in one but is missing from the other. However, some requirements should be taken to avoid the overuse of NIL structures such that the original structures must have common substructures but vary in the size for dissimilar substructures. For example, we can antiunify the two structures b and $f(a, b)$ through the application of NIL-theory by creating the term $nil(nil, b)$ which is $=_E$ to $f(b)$ and antiunifying $nil(nil, b)$ with $f(a, b)$ as depicted in Figure 3.8.

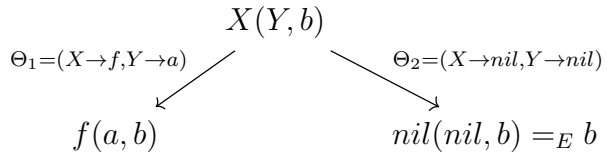

Figure 3.8: The antiunification of the terms f(a, b) and nil(nil,b).

We have also defined a set of equivalence equations to incorporate semantic knowledge of structural equivalences supported by the Java language specification as it provides various ways to define the same language specifications. These theories should be applies on higher-order ex-

tended structures to antiunify AST structures that are not identical but are semantically equivalent. For example, consider for- and while- statements that are two types of looping structure in Java programming language that have different syntax but semantically cover the same concept. Let us look at the **for**(i=0;i<10;i++) and **while**(i<10) code snippets, whose AST structures can be represented as **for**(initializer (i ,=,0) ,expression(i,<,10), updaters(i,++)) and **while**(expression(i ,<,10)), respectively. We could define an equivalence equation $=_E$ that allows the antiunification of for- and while- statements which are semantically similar structures. We also need to utilize the NIL-theory to handle varying number of arguments as the for- loop has three arguments whereas the while- loop has one. Using the NIL-theory we can create the structure **while**(nil (nil , nil , nil) ,expression(i,<,10), nil (nil , nil)) that is $=_E$ to **while**(expression(i,<,10)) and construct the antiunifier, V_0(V_1(V_2,V_3,V_4),expression(i,<,10), V_5(V_2,V_6)) as depicted in Figure 3.9.

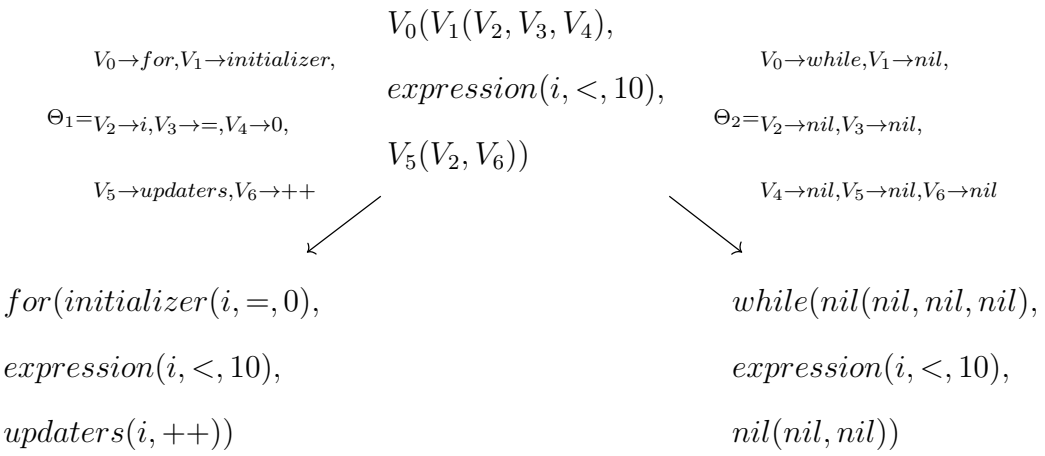

Figure 3.9: The antiunification of the structures **for**(initializer (i ,=,0) ,expression(i,<,10), updater(i,++)) and **while**(nil (nil , nil , nil) ,expression(i,<,10), nil (nil , nil)).

However, defining complex substitutions in higher-order antiunification modulo theories results in losing the uniqueness of MSA. For example, consider the terms $f(g(a, e))$ and $f(g(a, b), g(d, e))$. As described in Figure 3.10, two MSAs exist for these terms: we can antiunify $g(a, e)$ and $g(a, b)$ to create the antiunifier $g(a, X_0)$ and antiunify $g(d, e)$ with the NIL structure to create the antiunifier

$Y(Z, X_1)$; or we can antiunify $g(a, e)$ and $g(d, e)$ to create the antiunifier $g(X_0, e)$ and antiunify $g(a, b)$ with the NIL structure to create the antiunifier $Y(Z, X_1)$.

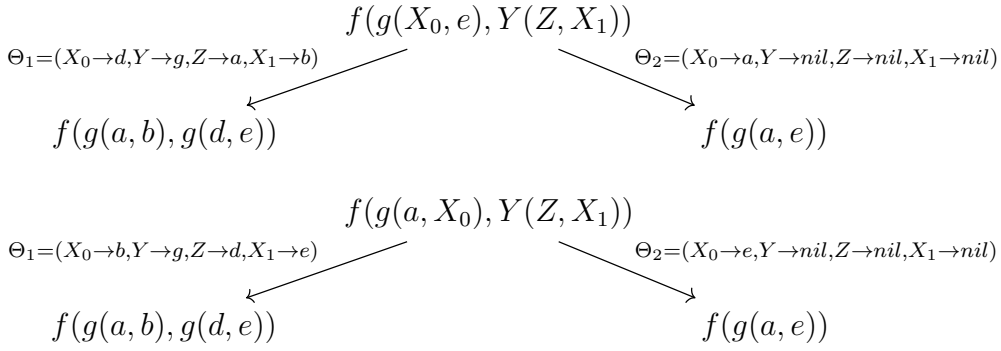

Figure 3.10: The antiunification of the terms $f(g(a, b), g(d, e))$ and $f(g(a, e))$ that creates multiple MSAs.

Despite having multiple potential MSAs, we need to determine one single MSA that is the most appropriate in our context. However, the complexity of finding an optimal MSA is undecidable in general [Cottrell et al., 2008] since an infinite number of possible substitutions can be applied on every variable. Therefore, we need to use an approximation technique to construct one of the best MSAs that can sufficiently solve our problem.

3.5 The Jigsaw framework

The Jigsaw tool is developed by Cottrell et al. [2008] to determine the structural correspondences between two Java source code fragments through the application of higher-order antiunification modulo theories such that one fragment can be integrated to the other one for small scale code reuse. Jigsaw could help us to determine potential candidate structural correspondences between AST nodes of logged Java classes by producing an augmented form of AST, called CAST (Correspondence AST), where each node holds a list of candidate correspondence connections between the two structures, each representing an antiunifier. It also develops a measure of similarity to indicate how similar the nodes involved in each correspondence connection are. The Jigsaw similarity

function relies on structural correspondence along with a simple knowledge of semantic equivalences supported by the Java language specification, and it returns a value between 0 and 1 that indicates zero and total structural matching, respectively. In addition, several semantical heuristics are used to improve the accuracy of similarity measurement by allowing the comparison of AST nodes that are not syntactically identical but are semantically related to each other.

As an example, the similarity between names of AST nodes is measured using a normalized computation based on the length of longest common substring. Another example is the comparison of **int** and **long** variable types, where an arbitrary value of 0.5 is defined as the similarity value as they are not syntactically identical but are not semantically unrelated. In addition, the Jigsaw framework also detects the structural correspondence between for-, enhanced-for-, while-, and do-loop statements; and if- and switch- conditional statements. As an example, Figure 3.11 shows the structural correspondence connections created by Jigsaw between the AST nodes of Examples 1 and 2 along with the similarity value for each correspondence connection.

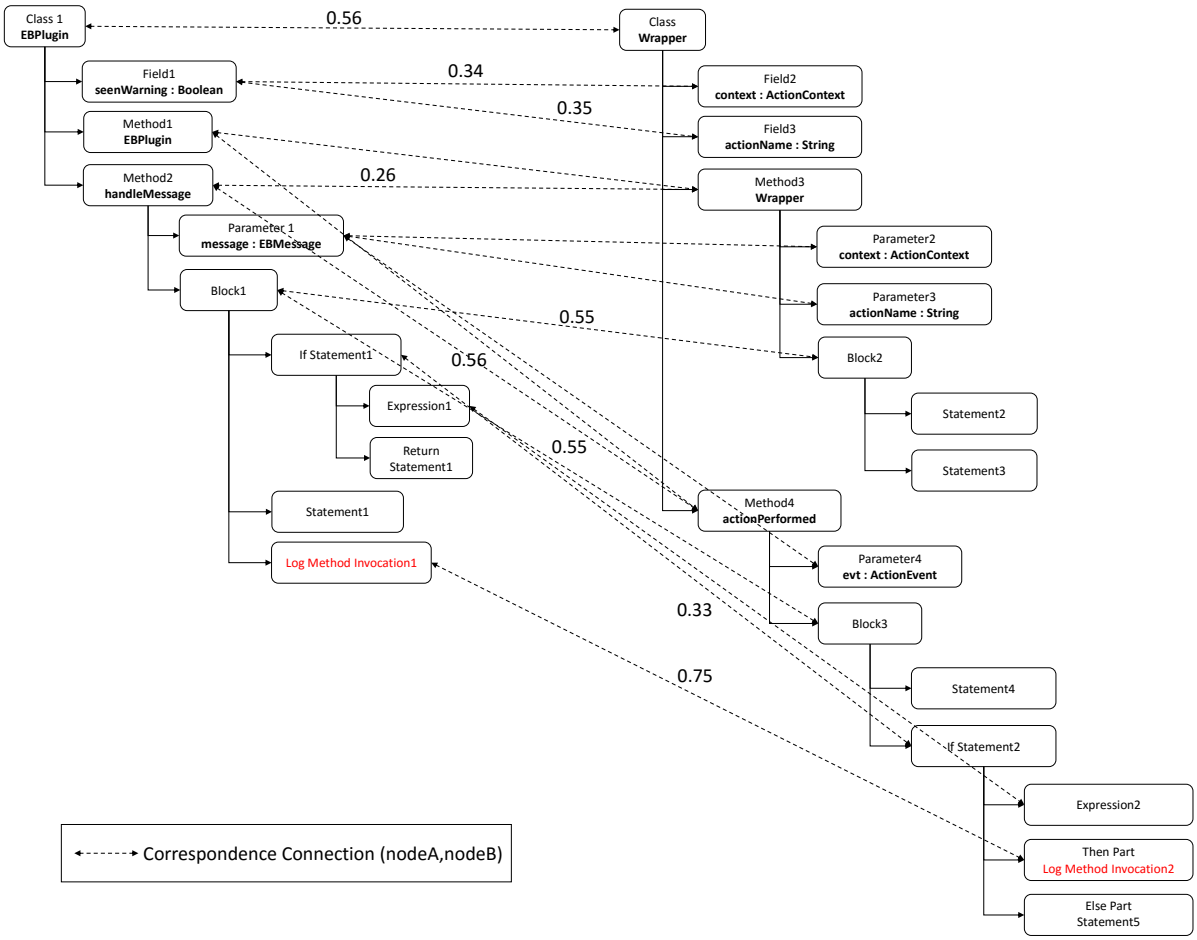

Figure 3.11: Simple CAST structure of examples in Figures 3.4 and 3.5. The links between AST nodes indicate structural correspondence connections created by the Jigsaw framework along with the similarity value.

However, the Jigsaw tool does not suffice to construct an antiunifier that is the best fit to our application. In addition, the Jigsaw similarity function does not measure the similarity of two logged Java classes with a focus on logging calls, which is needed in our context. To address these issues, we should develop a greedy selection algorithm to approximate the best antiunifier by determining the best correspondence for each node. In the following chapter, we will discuss our approach to construct structural generalizations and our implementation by means of the higher-order antiunification modulo theories and the Jigsaw framework.

3.6 Summary

In this chapter, we described antiunification as a technique to construct a common generalization of two given terms. We have also introduced an extended form of antiunification, which is called higher-order antiunification modulo theories, where a set of equivalence equations can be applied on higher-order extended structures to incorporate background knowledge. In addition, we provided a brief description of AST that maps Java source code in a tree structure form, and why an extended form of it, named AUAST, is required to create higher-order structures specific to our problem context. Finally, we discuss the Jigsaw framework and how it could assist us in determining the potential structural correspondences.

Chapter 4

Antiunification of Logged Java Classes

In Chapter 3 we provided background information on higher-order antiunification modulo theories—a theoretical framework for constructing a generalization from two given structures—and we described how the Jigsaw tool applies this framework on AST structures of two given Java classes to determine potential structural correspondences between them. We now consider how these frameworks could help us (1) to generalize ASTs of two Java classes containing logging calls and (2) to develop a similarity measure with a focus on logging calls that can provide us with useful information for clustering LJCs in a later phase.

Recall the general point of this study: we aim to provide a concise description of where logging calls happen in the source code through constructing structural generalizations that represent the detailed structural similarities and differences of LJCs. To this end, we should develop an algorithm that:

- classifies LJCs into groups using a measure of similarity such that entities in each group has maximum similarity with each other and minimum similarity to other ones
- abstracts structural correspondences of LJCs of each group into a structural generalization representing the similarities and differences

To construct structural generalizations from a set of LJCs, we developed a prototype tool that applies the Jigsaw framework to find candidate correspondences between two ASTs, the HOAUMT to generalize the structures, and a modified version of the agglomerative hierarchical clustering algorithm to classify a set of LJCs using a measure of similarity. As explained in Section 3.3, the AST structure should be extended to AUAST structure that allows the insertion of variables in place of any node, which is required for HOAUMT.

Our hierarchical clustering algorithm is a bottom-up approach that starts with singleton clusters, where each contains one AUAST. In every iteration, it merges the closest clusters which are the clusters with maximum similarity between their AUASTs. Therefore, we need to develop a measure of similarity between each pair of AUAST and then construct an antiunifier when it is needed to merge two clusters.

The structural similarity between two given AUASTs is defined as the number of identical simple property values over total number of simple property values of the antiunifier (see Section 4.5). To do so, we determine the best correspondences for each node and compute the structural matches between them. Our tool performs a sequence of 3 actions to determine the best correspondences between two AUASTs, outlined by the algorithm DETERMINE-BEST-CORRESPONDENCES: (1) it generates all possible candidate correspondence connections between ASTs of two AUASTs using the Jigsaw framework (line 1) (see Section 4.2); (2) it applies some constraints to prevent the antiunification of logging calls with anything else (line 2) (see Section 4.3); (3) it determines the best correspondence for each node of AUASTs with the highest similarity and then removes the other correspondence connections involving those nodes (line 3) (see Section 4.4);

To construct an approximation of the best antiunifier to our problem with a special attention to logging calls, a further step should be taken, which is antiunification of each AUAST node with its best correspondence determined in the previous step through antiunifying their structural properties (see Section 4.6). Figure 4.1 shows an overview of the general process of our antiunification technique, as will be described in the following sections.

Algorithm 4.1 DETERMINE-BEST-CORRESPONDENCES($auastA$,$auastB$) determines best correspondences between the two AUAST nodes $auastA$ and $auastB$

DETERMINE-BEST-CORRESPONDENCE($auastA$**,** $auastB$**)**
1: JIGSAW-CORRESPONDENCE($auastA$, $auastB$)
2: APPLY-CONSTRAINS($auastA$, $auastB$)
3: DETERMINE-CORRESPONDENCES($auastA$)

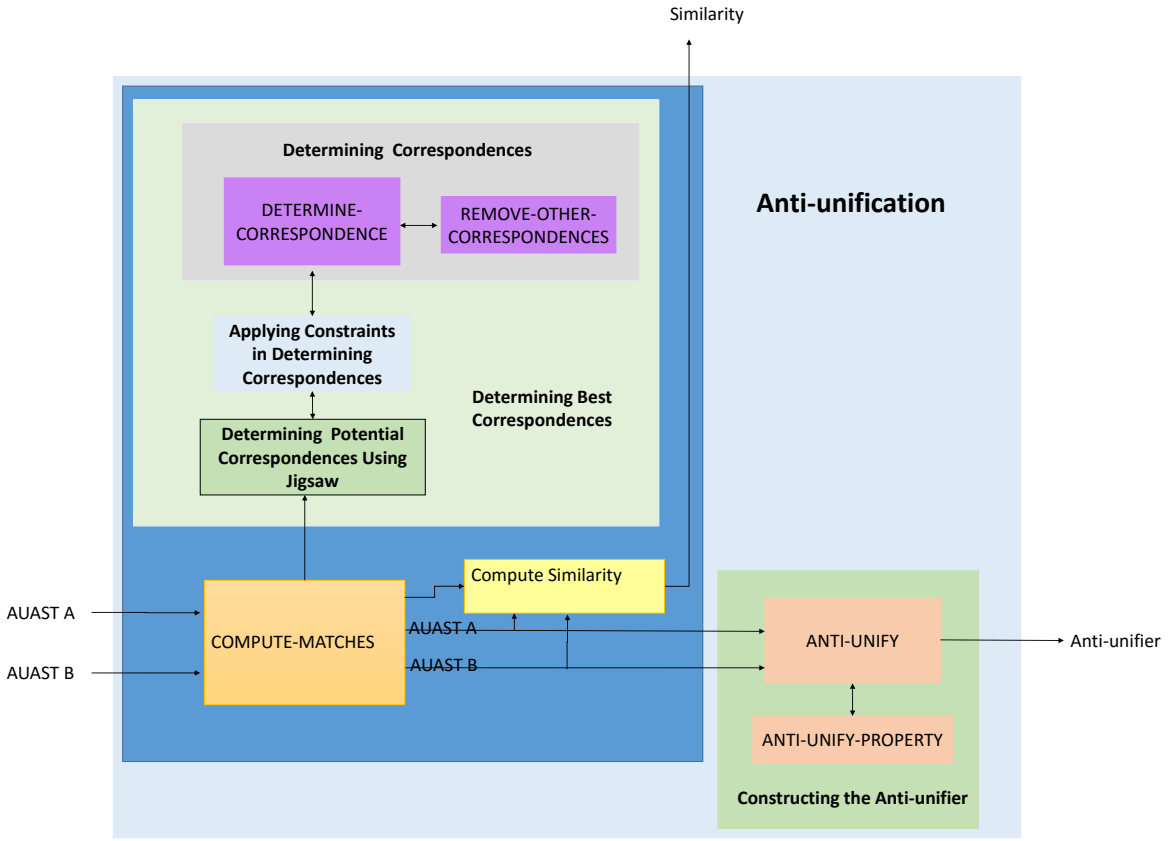

Figure 4.1: Overview of the system.

4.1 Constructing the AUAST

The goal of this phase is to construct an extension of the AST structure that would allow the creation of an antiunified structure. As described in Section **??**, an antiunified structure utilizes variables that must be substituted with a proper substructure to gain back to each original structure; however, the AST structure does not contain any variables thus an extended form of it is required, named AUAST, to address these limitation by allowing the insertion of variables in place of any node in the tree structure, including both subtrees and leaves, to indicate variations between original structures. the AUAST structure addresses the limitations of AST to construct an antiunifier

by adding the following structural properties:

- **Simple Variable Property**: an extension of simple property referring to two simple property values to allow the insertion of variables in place of leaves.

- **Child Variable Property**: an extension of child property referring to two child nodes to allow the insertion of variables in place of subtrees.

We provide an example to demonstrate the AUAST structure, which is limited to log method invocation subtrees of the sample Java classes shown in Figure 4.2. The log method invocation nodes both contains `EXPRESSION`, `ARGUEMENTS`, and `NAME` structural properties which are made up of **`Log`**, **`Log`**, **`WARNING`** simple values for the AUAST1 and **`Log`**, **`Log`**, **`ERROR`** simple values for the AUAST2, respectively. The structural representation of the AUASTs as defined in Section **??** is `EXPRESSION[EXPRESSION[IDENTIFIER[`**`Log`**`]], ARGUMENTS[QUALIFIER[IDENTIFIER[`**`Log`**`]], NAME[IDENTIFIER[`**`WARNING`**`]]` for the AUAST1 and `EXPRESSION[EXPRESSION[IDENTIFIER[`**`Log`**`]], ARGUMENTS[QUALIFIER[IDENTIFIER[`**`Log`**`]], NAME[IDENTIFIER[`**`ERROR`**`]]` for the AUAST2, where the words capitalized represents subtrees and the words shown in bold represents leaves of the tree structure.

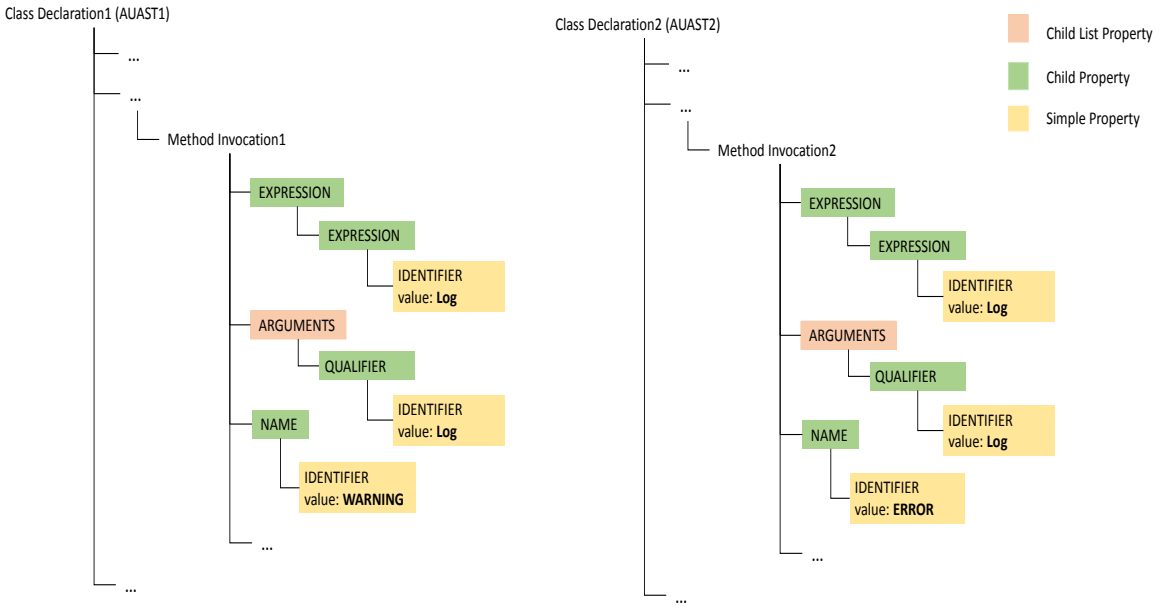

Figure 4.2: The AUASTs of log Method Invocation nodes from the Java classes in Figure 3.4 and Figure 3.5.

4.2 Determining Correspondences Using Jigsaw

Algorithm 4.2 JIGSAW-CORRESPONDENCE($auastA$, $auastB$) determines all the potential correspondences between nodes of two given AUASTs

JIGSAW-CORRESPONDENCE($auastA$, $auastB$)
1: **for do** $astA \in auastA$
2: **for do** $astB \in auastB$
3: $castA, castB \leftarrow$ JIGSAW-ANTIUNIFY$(astA, astB)$
4: **end for**
5: **end for**

4.3 Constraints in Determining Correspondences

To construct an antiunifier of two AUASTs with a focus on logging calls, some constraints should be applied prior to determining the best correspondences. The first constraint (as described below)

should be applied to prevent the antiunification of log method invocation nodes with any other type of node.

Constraint 1. A logging call should either be antiunified with another logging call or should be antiunified with "nothing".

This constraint creates a further constraint, which is:

Constraint 2. A structure containing a logging call should be antiunified with a corresponding structure containing another logging call or should be antiunified with "nothing".

To provide an example to illustrate it consider ASTs of two Java classes in Figure 3.11. Jigsaw creates a correspondence connection between the two log method invocation nodes and the two **if** statements. As is clear, the second **if** statement contains a logging call, while there is no corresponding logging call in the first one. According to the first constraint, two log method invocation nodes should be antiunified together. On the other hand, a correspondence connection is created between the two **if** statements; however, antiunification of these statements includes antiunifying their children nodes as well. Thus, statements inside the body of **if** statements must be antiunified with each other, indicating that log method invocation inside the body of **if** statement in the second example should be antiunified with "nothing", which is contrary to our first assumption. In order to comply with the first constraint, the correspondence connection between two **if** statements should be deleted, leading us to apply the second constraint.

Our approach applies these constraints by taking the following steps prior to determining correspondences:

1. Augment a property to AUAST node to mark log method invocation nodes and structures enclosing them as "logged".

2. Remove correspondence connections where one node is marked as "logged" and the corresponding node is not.

4.4 Determining Correspondences

As explained in Section 4.2, each node of the AUAST structure holds a list of candidate correspondence connections where each represents an antiunifier. Despite having multiple potential antiunifiers, we need to determine one single antiunifier that is helpful to solve our problem. In general, higher order antiunification modulo theories is undecidable [Cottrell et al., 2008]. That is, the complexity of determining the most optimal MSA is undecidable, but our desire is to create one of the best MSAs to approximate the optimal one that can sufficiently solve our problem, thus the antiunification process should construct an antiunifier that is the best approximate fit for our application. To this end, a greedy selection algorithm has been used, which is an approximation technique to determine the best correspondence for each node in the AUAST so constructing the antiunifier that is approximately the best fit to our problem. As a result, each node can either be antiunified with its best correspondence in the other AUAST or with "nothing".

DETERMINE-CORRESPONDENCE algorithm greedily selects the most similar correspondence as the best fit for each node in AUAST. It takes one of the AUASTs, visiting the AUAST nodes therein to store all candidate correspondence connections between the two AUAST nodes in a list, which is sorted in a descending order based on the Jigsaw similarity measure (lines 1–8). The correspondence connection with the highest similarity value is determined as the best fit for the two nodes involved (lines 9–11); all other correspondence connections involving these two nodes are removed using REMOVE-OTHER-CORRESPONDENCES algorithm (line 10). This process terminates when no more correspondence connections is left in the list.

REMOVE-OTHER-CORRESPONDENCES algorithm removes correspondence connections that are not selected as the best fit from three lists: the list of all correspondence connections (Line 5 and Line 12); the list of candidate correspondence connections of the first node involved in these connections(Line 6 and Line 13); the list of candidate correspondence connections of the second node involved in these connections(Line 7 and Line 14).

As an example, Figure 4.3 shows the correspondences between AUAST nodes after applying

Algorithm 4.3 DETERMINE-CORRESPONDENCE($auastA$) takes in an AUAST node and create a list of correspondence connections containing the best correspondence to each node in the AUAST.

DETERMINE-CORRESPONDENCE($auastA$)

```
1: list ← ()
2: nodes ← VISITOR(auastA)
3: for donode ∈ nodes
4:     for doce ∈ correspondences[node]
5:         APPEND(ce, list)
6:     end for
7: end for
8: SORT(list)
9: for ce ∈ list do
10:    REMOVE-OTHER-CORRESPONDENCES(ce, list)
11: end for
12: return list
```

Algorithm 4.4 REMOVE-OTHER-CORRESPONDENCES(ce, $list$) Remove all other correspondences involving nodes of a particular correspondence connection or element (ce) from lists of correspondence connections.

REMOVE-OTHER-CORRESPONDENCES(ce, $list$)

```
1: list1 ← correspondences[nodeA[ce]]
2: list2 ← correspondences[nodeB[ce]]
3: for ce1 ∈ list1 do
4:     if ce1 ≠ ce then
5:         REMOVE(ce1, list)
6:         REMOVE(ce1, correspondences[nodeA[ce1]])
7:         REMOVE(ce1, correspondences[nodeB[ce1]])
8:     end if
9: end for
10: for ce2 ∈ list2 do
11:    if ce2 ≠ ce then
12:        REMOVE(ce2, list)
13:        REMOVE(ce2, correspondences[nodeA[ce2]])
14:        REMOVE(ce2, correspondences[nodeB[ce2]])
15:    end if
16: end for
```

the constraints and DETERMINE-BEST-CORRESPONDENCE algorithm on the list of correspondence connections created by the Jigsaw framework in Figure 3.11.

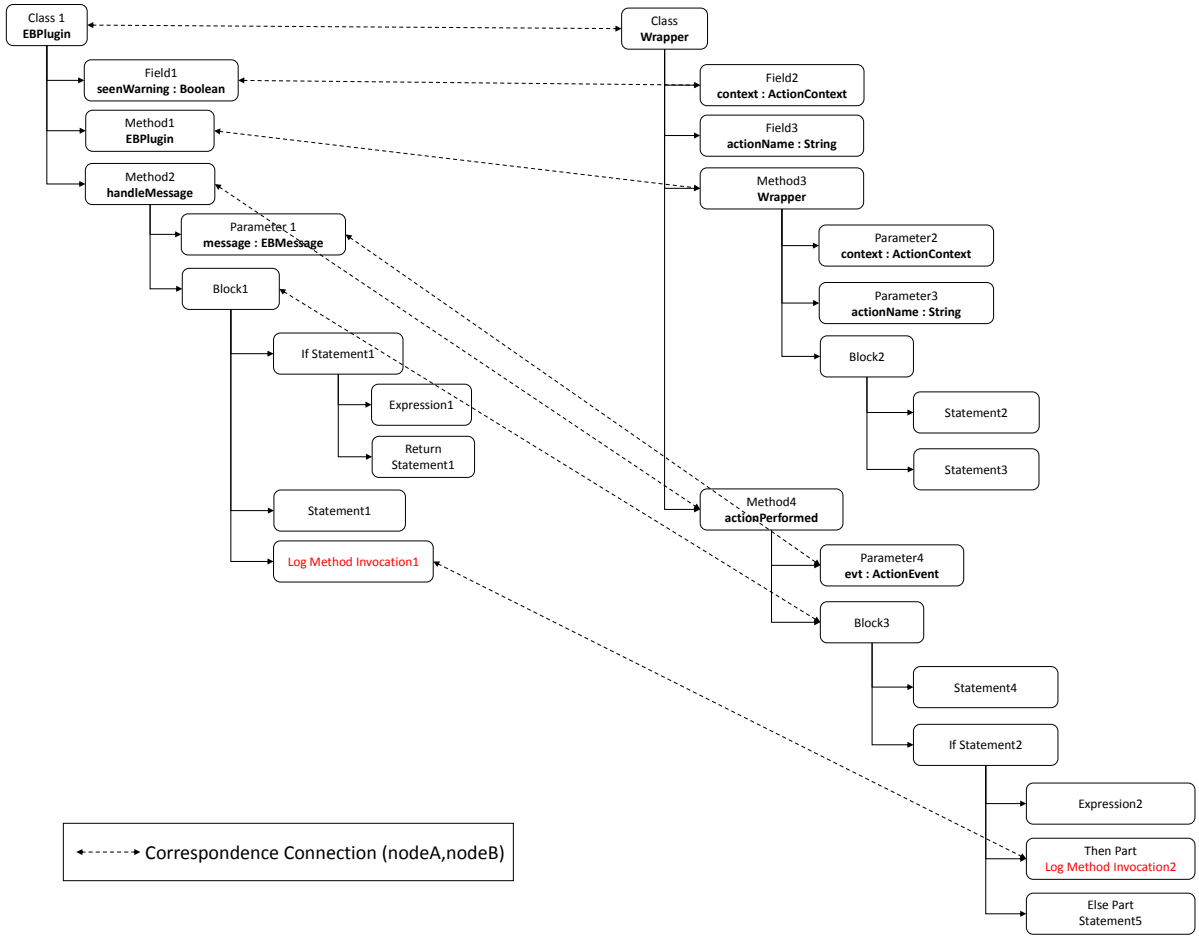

Figure 4.3: Simple AUAST structures constructed from the ASTs in Figure 3.11. Links between AUAST nodes indicate structural correspondences selected as the best fit

4.5 Computing Similarity

Similarity computation is particularly important for the clustering phase that relies on accurate estimation of distance between logged Java classes. The notion of similarity can differ depending on the given context. That is, similarity between certain features could be highly important for a particular application, while it is not for the other one. The utility of a similarity function can

be determined based on how good it enables us to produce accurate results for a particular task. In this study, a similarity measure is needed to classify Java classes that use logging calls based on structural similarity between them. The structural similarity of two AUASTs can be defined as the number of identical simple structural property values over total number of simple structural property values of the antiunifier.

Algorithm 4.5 COMPUTE-MATCHES($auastA$, $auastB$), determines the matches between two AUASTs via a recursive traversal of structural properties

```
    COMPUTE-MATCHES(auastA, auastB)
 1: if auastB ≠ NULL then
 2:     DETERMINE-BEST-CORRESPONDENCE(auastA, auastB)
 3: end if
 4: matches ← 0
 5: for property ∈ properties[ant − unifier] do
 6:     valueA ← value[property]
 7:     valueB ← value[GETCORRESPONDENCE(valueA)]
 8:     if property instanceof SimpleProperty or property instanceof SimpleVariableProperty
    then
 9:         matches ← matches + JIGSAW-MATCHES(valueA, valueB)
10:     else if property instanceof ChildProperty or property instanceof ChildVariableProperty
    then
11:         matches ← matches + COMPUTE-MATCHES(valueA, valueB)
12:     else if property instanceof ChildListProperty then
13:         for nodeA ∈ valueA do
14:             nodeB ← GETCORRESPONDENCE(nodeA)
15:             matches ← matches + COMPUTE-MATCHES(nodeA, nodeB)
16:         end for
17:     end if
18: end for
19: return matches
```

The number of matches between $auastA$ and $auastB$ is computed via the COMPUTE-MATCHES algorithm through a recursive traversal of structural properties of the nodes. First, the best correspondences are selected using the DETERMINE-BEST-CORRESPONDENCE algorithm. For simple and simple variable structural properties, the number of matches is computed re-using the Jigsaw similarity function that computes the number of matches between the property values(Lines 8-9). For child and child variable structural properties, the number of matches is computed recursively

for the child node and is propagated to the parent(Lines 10-11). For child list structural properties, the number of matches is computed for each child node recursively and is propagated to the parent node(Lines 12-17). All matches are summed up to compute total number of matches between the two AUASTs. Then the following equation is used to compute the structural similarity between $auastA$ and $auastB$:

$$similarity = \frac{2 * matches}{|auastA| + |auastB|} \tag{4.1}$$

Where total number of simple values for $auastA$ and $auastB$ is computed via COMPUTE-MATCHES($auastA$) and COMPUTE-MATCHES($auastB$), respectively. The similarity function returns a value between 0 and 1 where indicate zero and total class matching, respectively.

4.6 Constructing the antiunifier

Once the best correspondences has been determined between AUAST nodes, we construct a new antiunified AUAST by traversing AUAST structures recursively and antiunifying the structural properties. The new antiunified structure is a generalization of two original structure, called antiunifier, where common structural properties are represented by copy, and differences in structural properties are represented by structural variables. The variables may be inserted in place of any node in AUAST including both subtrees and leaves and can be substituted with proper original substructures to gain back to original structures.

Antiunification of two AUAST nodes is performed through antiunification of their structural properties, via the ANTIUNIFY algorithm. For each structural property of $auastA$ and $auastB$, where there is no corresponding property in the other AUAST, a structural variable property is created through antiunifying the structural property with the NIL structure via the ANTIUNIFY-PROPERTY algorithm and added to properties of the antiunifier (Lines 3-6 and Lines 13-17); if both nodes has the same property but with different property values, a structural variable property is created via the ANTIUNIFY-PROPERTY algorithm and appended to the antiunifier structural

properties (Lines 7-8); otherwise, if the two nodes has the same exact structural property, a copy of one of them is added to the antiunifier structural properties (Lines 10-11).

Algorithm 4.6 Input into ANTIUNIFY($auastA$, $auastB$) are two AUAST nodes. This algorithm construct an antiunified AUAST node through antiunification of input node's structural properties.

```
    ANTIUNIFY(auastA, auastB)
 1: antiunifier ← Null
 2: for propA ∈ properties[auastA] do
 3:     valueA ← value[property]
 4:     if CONTAINS(auastB, propA) = NULL then
 5:         ADDPROPERTY(antiunifier, ANTIUNIFY-PROPERTY(propA, NIL))
 6:     else if valueA ≠ value[CONTAINS(auastB, propA)] then
 7:         ADDPROPERTY(antiunifier, ANTIUNIFY-PROPERTY(propA, CONTAINS(auastB, propA))
 8:     else
 9:         ADDPROPERTY(antiunifier, propA)
10:     end if
11: end for
12: for propB ∈ properties[auastB] do
13:     if CONTAINS(auastA, propB) = NULL then
14:         ADDPROPERTY(antiunifier, ANTIUNIFY-PROPERTY(propB, NIL))
15:     end if
16: end for
17: return antiunifier
```

Antiunification of structural properties $propA$ and $propB$ is performed via the ANTIUNIFY-PROPERTY algorithm. If $propA$ is a simple property, a simple variable property is constructed referring to two simple values (Lines 2-3); If structural property is a child property, a child variable structure is constructed (Line 5); if structural property is a child list property, for each child of $propA$ and $propB$, where there is no correspondence in the other AUAST, an antiunified node is created through antiunifying the child node with the NIL structure via ANTIUNIFY algorithm and added to the value of the antiunified child list property; otherwise, the child node is antiunified with its best correspondence (Lines 6-18).

For example, we supply ANTIUNIFY algorithm with the log method invocation nodes from the AUASTs in Figure 4.2. `EXPRESSION` and `ARGUMENTS` are similar in both AUASTs thus a copy of them will be added to structural properties of the antiunified AUAST (Line 10); however, the simple value of Name property is different in both structures thus a call to ANTIUNIFY-

Algorithm 4.7 ANTIUNIFY-PROPERTY($propA$, $propB$) takes two structural properties and creates an antiunified structural property.

```
    ANTIUNIFY-PROPERTY(propA, propB)
 1: property ← Null
 2: if propA instanceof SimpleProperty then
 3:     property ← CREATE-SIMPLE-VARIABLE-PROPERTY(propA, propB)
 4: else if propA instanceof ChildProperty then
 5:     property ← CREATE-CHILD-VARIABLE-PROPERTY(propA, propB)
 6: else if propA instanceof ChildListProperty then
 7:     for child ∈ value[propA] do
 8:         if correspondence[child] ≠ NULL then
 9:             APPEND(children, ANTIUNIFY(child, correspondence[child]))
10:         else
11:             APPEND(children, ANTIUNIFY(child, NIL))
12:         end if
13:     end for
14:     for child ∈ value[propB] do
15:         if correspondence[child] = NULL then
16:             APPEND(children, ANTIUNIFY(child, NIL))
17:         end if
18:     end for
19:     value[property] ← children
20: end if
21: return property
```

PROPERTY on Line 8 will return a simple structural variable. Figure 4.4 shows the antiunified AUAST, where the annotation `WARNING-or-ERROR` is used to represent the simple structural variable that must be substituted with either `WARNING` or `ERROR` simple value to gain back to each original AUAST structure. The structural representation of the antiunified AUAST is `EXPRESSION[EXPRESSION[IDENTIFIER[`**`Log`**`]], ARGUMENTS[QUALIFIER[IDENTIFIER[`**`Log`**`]], NAME[IDENTIFIER[`**`WARNING-or-ERROR`**`]].`

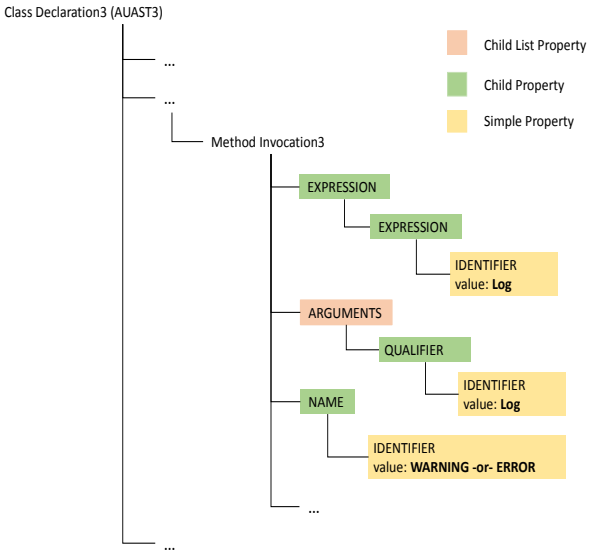

Figure 4.4: The antiunifier (AUAST3) constructed from log Method Invocation AUAST nodes in Figure 4.2

Figure 4.5 shows a simple view of the antiunified AUAST constructed from the two AUASTs in Figure 4.3, where "$a \langle\rangle b$" represents that the two subtrees a and b are antiunified with each other in the antiunifier and "a-or-b" represents a simple structural variable that must be substituted with either a or b simple value to recover each original structure.

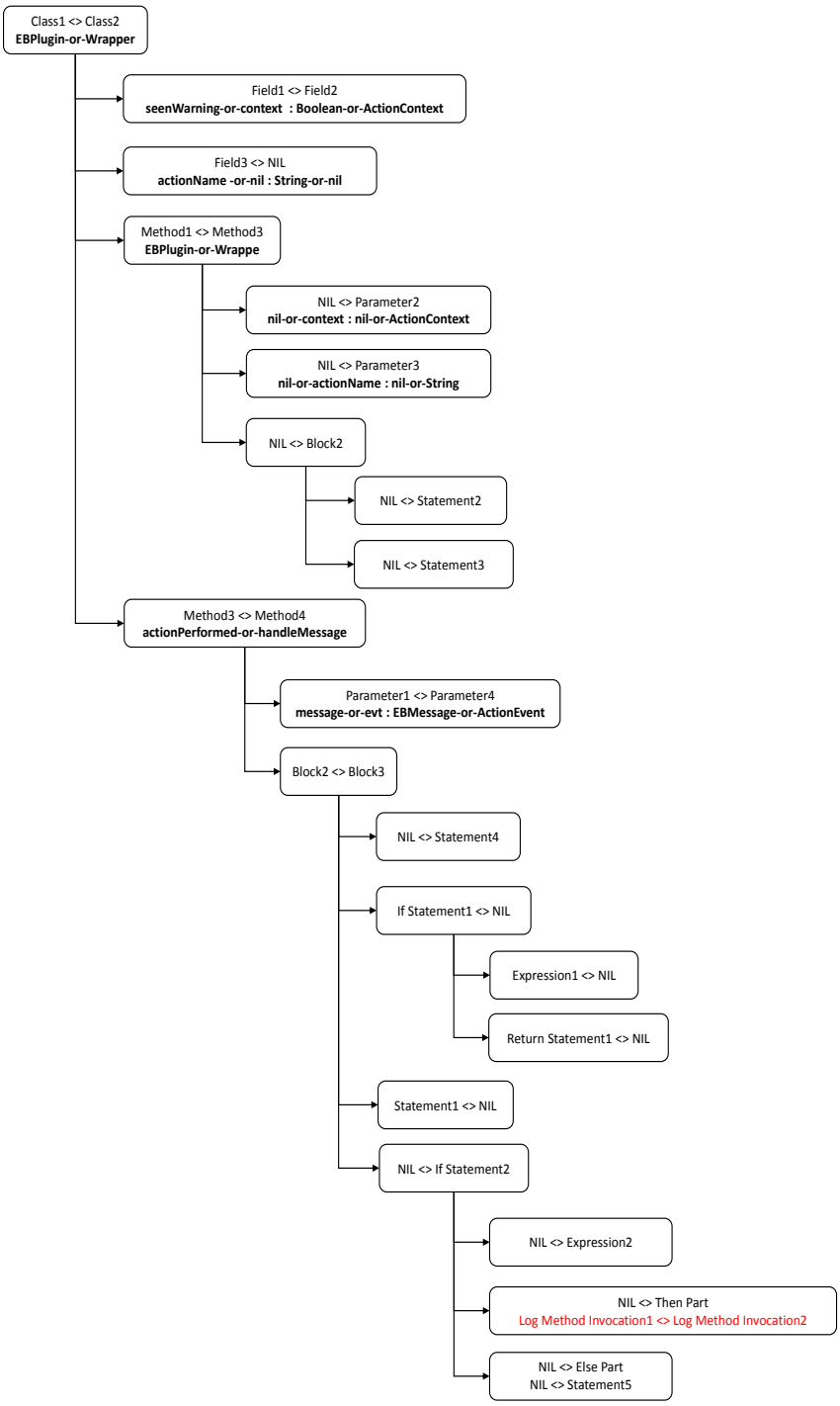

Figure 4.5: Simple antiunified AUAST structure of the two AUASTs in Figure 4.3

4.7 Multiple logging calls

Problem: There might be some cases that our approach is not able to antiunify logging calls in two input seeds, when there is more than one logging call in a logged Java class. For example, consider the logged Java classes in Figures 4.6 and 4.7. Figure 4.8 shows the simple AUASTs for these examples and all potential correspondence connections between the AUAST nodes. Figure 4.9 shows the correspondence connections selected as the best match using our greedy algorithm. To antiunify method1 with method3, we should antiunify their structural properties; thus, log1 should be antiunified with log3 and log4 should be antiunified with "nothing" since there is no corresponding logging call in the body of method1, while there is a corresponding logging call for log4 in the body of method2 (log2).

```
1 public class test1{
2         public void method1(){
3                 ...
4                 Log.log();
5                 ...
6         }
7         public void method2(){
8                 ...
9                 Log.log();
10                ...
11        }
12 }
```

Figure 4.6: A Java class that utilizes multiple logging calls. This will be referred to as Example 1.

```
1 public class test2{
2         public void method3(){
3                 ...
4                 Log.log();
5                 ...
6                 Log.log();
7         }
8 }
```

Figure 4.7: A Java class that utilizes multiple logging calls. This will be referred to as Example 2.

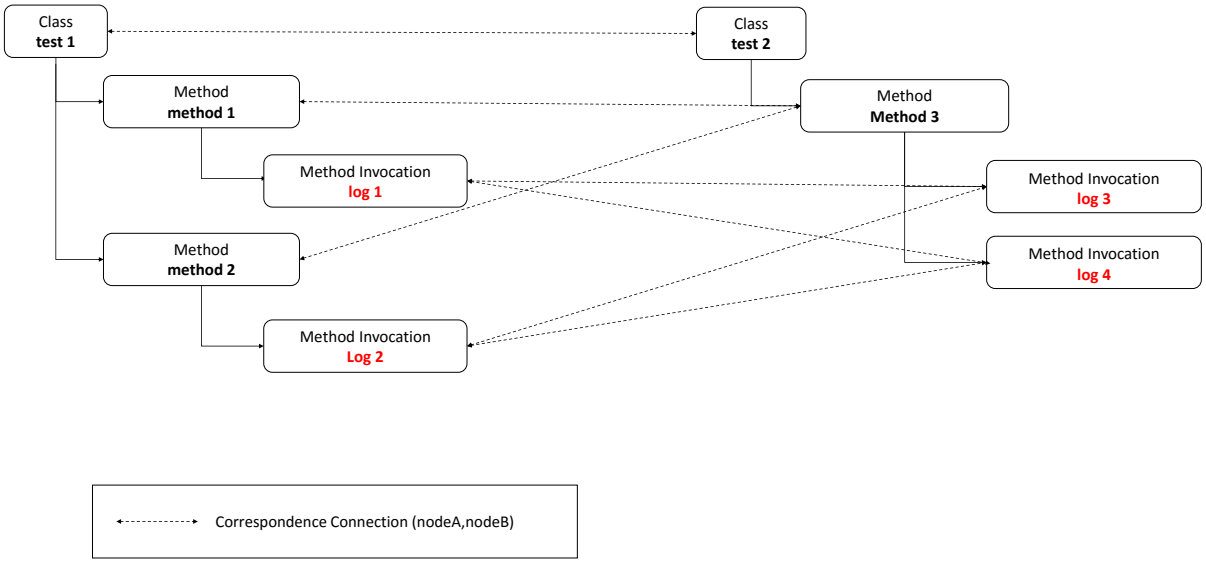

Figure 4.8: Simple AUAST structure of examples in Figures 4.6 and 4.7. Links between AUAST nodes indicate potential candidate structural correspondences detected by the Jigsaw framework.

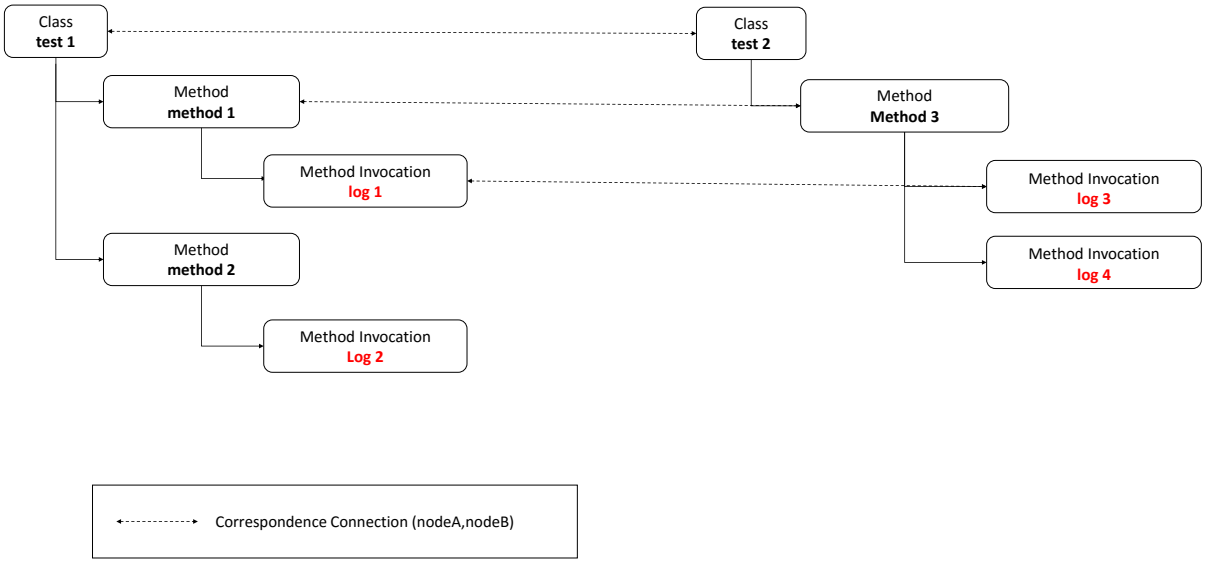

Figure 4.9: Simple AUAST structure of examples in Figures 4.6 and 4.7. Links between AUAST nodes indicate structural correspondences selected as the best match using our greedy algorithm.

Suggested Solution: We can split these cases into more than one case, where each logged Java class contains only one logging call. To do so, we need to create a copy of logged Java class for each logging call by maintaining the logging call and removing the other ones. For example, we need to create two copies for each logged Java class of Examples 1 and 2 as depicted in Figures 4.10 and 4.11, respectively.

```
1  public class test1{
2          public void method1(){
3                  ...
4                  Log.log();
5                  ...
6          }
7          public void method2(){
8                  ...
9                  //removed
10                 ...
11         }
12 }
13
14 public class test1{
15         public void method1(){
16                 ...
17                 //removed
18                 ...
19         }
20         public void method2(){
21                 ...
22                 Log.log();
23                 ...
24         }
25 }
```

Figure 4.10: Create multiple copies of Example 1 for each logging call.

```
1 public class test2{
2         public void method3(){
3                 ...
4                 Log.log();
5                 ...
6                 //removed
7         }
8 }
9
10 public class test2{
11        public void method3(){
12                ...
13                //removed
14                ...
15                Log.log();
16        }
17 }
```

Figure 4.11: Create multiple copies of Example 2 for each logging call.

4.8 Antiunifying a set of AUASTs

- PROBLEM: antiunifying a set of AUASTs of LJCs
- SOLUTION: Developing a modified version of a hierarchical agglomerative clustering algorithm (illustrated in Figure 4.12) as described below:

  1. Start with singleton clusters, where each cluster contains one AUAST
  2. Compute the similarity between clusters in a pairwise manner
  3. Find the closest clusters (a pair of clusters with maximum similarity)
  4. Merge the closest cluster pair and replace them with a new cluster containing antiunifier of AUASTs of the two clusters
  5. Compute the similarity between the new cluster and all remaining clusters

  – Repeat Steps 3,4, and 5 until the similarity between closest clusters becomes below a predetermined threshold value

- The similarity between a pair of clusters is defined as the similarity between their AUASTs
- Determine the similarity threshold value through informal experimentation

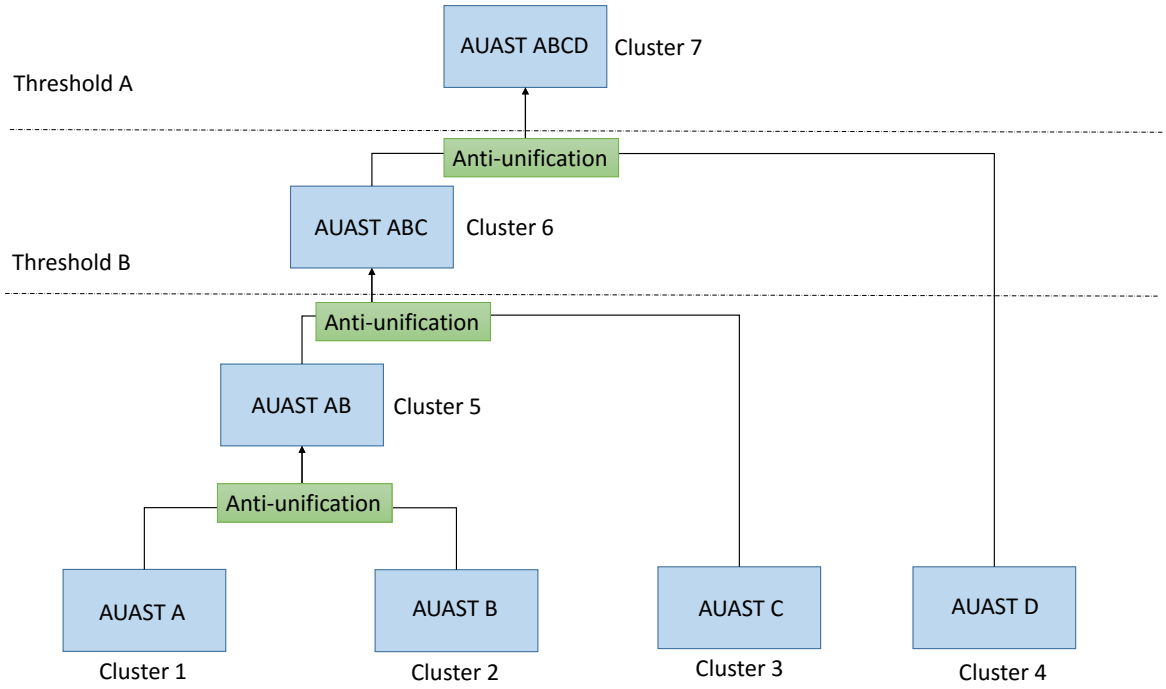

Figure 4.12: Antiunification of 4 AUAST nodes using an agglomerative hierarchical clustering algorithm. The threshold value indicates the number of clusters we will come up with.

Chapter 5

Evaluation

- Two empirical studies were conducted

5.1 Experiment 1

- First experiment is conducted to evaluate the accuracy of our approach and tool
- It addresses the following research questions:
  - RQ1: can our tool determine the structural similarities and differences between logged Java classes correctly?
  - RQ2: can our tool compute the similarity between logged Java classes correctly?
- To do so, 10 logged Java classes were selected randomly from jEdit v4.2 pre 15 (2004), as our test set
- We apply our tool on the test set to create generalizations and to compute the similarity value between logged Java classes in a pairwise manner
- To address the first research question we compute the following measurements for each test case:
  - the number of correspondences that our tool detects correctly
  - the total number of correspondences
- To determine the correct correspondences we performed a manual investigation
- To address the second research question we compute:

  - the number of similarity values between logged Java classes that are computed correctly by our tool
  - the total number of comparisons

- The correct similarity value for each comparison is calculated manually by
- The results taken form our tool are compared with the results taken by manual investigation using the JUnit testing framework

5.2 Experiment 2

- Second experiment is conducted to address the following research questions:
  - RQ3: what structural similarities and differences do logged Java classes have?
  - RQ4: Is it possible to find common patterns in where logging calls do occur?
- To do so, we applied our tool on the source code of three open-source full systems that make use of logging to determine the patterns on a per-system class-granularity basis analysis
- These systems are different from the system that the test set is selected from

5.3 Results

- I will describe the results taken form the second experiment

5.4 Lessons learned

- I will describe our findings

Chapter 6

Discussion

6.1 Threats to validity

- Our goal is to recognize the limitations and pitfalls of our approach and its developed tool support
- The first potential thread to validity of our characterization study is the degree to which our sample set of software systems is a good representation of all real-world logging practices. To address this issue we selected software systems that:
  - are different in terms of application
  - are among the most popular applications in their own product category
  - has long history in software development
- Secondly, our manual investigation to find the correct correspondences might be biased due to human errors. To limit the bias
  - other people can be involved to double check the accuracy of manual work in the future work
- However, these results are still promising

6.2 Our tool output

- We investigated the cases where our tool fails and we found that the failures are due to:
  - the assumptions taken in developing the algorithms
  - the fundamental limitations and complexities in determining the detailed structural similarities and differences

- The are some issues that our tool is not able to handle perfectly during generalization:
  - maintaining the correct ordering of statements inside the method bodies
  - resolving all the conflicts that happen in determining the best correspondences
  - producing executable generalizations

6.3 Theoretical foundation

- antiunification and its extensions has several theoretical and practical applications:
  - analogy making [Schmidt, 2010]
  - determining lemma generation in equational inductive proofs [Burghardt, 2005]
  - detecting the construction laws for a sequence of structures [Burghardt, 2005]
- Using higher-order antiunification modulo theories in our application, which is undecidable in general, leads us to take approximations suitable to our context
- The set of equational theories should be developed particularly for the structure used in each problem context

Chapter 7

Related Work

In this chapter, we review related work to the topics of our study including: the application of logging in real-world software systems (Section 7.1), determining correspondences in the source code (Section 7.2), data mining approaches to extract API usage patterns (Section 7.3), antiunification and its application to detect strcutural correspondences and construct generalizations (Section 7.4), and clustering (Section 7.5).

7.1 Usage of logging

Logging is a conventional programming practice to record a software system's runtime information that can be used in post-modern analysis to trace the root causes of systems' activities. Log analysis is most often performed for failure diagnosis, system behavioral understanding, system security monitoring and performance diagnostics purposes as described below:

- **Log analysis for failure diagnosis:** Xu et al. [2009] use statistical techniques to learn a decision tree based signature from the console logs and then utilize the signature to diagnose anomalies. SherLog [Yuan et al.] uses failure log messages to infer the source code paths that might have been executed during a failure.

- **Log analysis for system behavior understanding:** Fu et al. [2013] present an approach for understanding system behavior through contextual analysis of logs. They first extracted execution patterns reflected by a sequence of system logs and then utilized the patterns to find contextual factors from logs that causes a specific system behavior. The Linux Trace Toolkit [Yaghmour and Dagenais, 2000] was created to record and analyze system behavior by providing an efficient kernel-level event logging infrastructure. A

more flexible approach is taken by DTrace [Cantrill et al., 2004] which allows dynamic modification of kernel code.

- **Log analysis for system security monitoring:** Bishop [1989] proposes a formal model of system's security monitoring using logging and auditing. Peisert et al. [2007] have developed a model that demonstrates a mechanism for extracting logging information to detect how an intrusion occurs in software systems.

- **Log analysis for performance diagnosis:** Nagaraj et al., [2012] developed an automated tool to assist developers in diagnosis and correction of performance issues in distributed systems by analyzing system behaviors extracted from the log data.

Jiang et al. [2009a] study the effectiveness of logging in problem diagnosis. Their study shows that customer problems in software systems with logging resolve faster than those without logging by investigating the correlations between failure root causes and diagnosis time. Despite the importance of logging for software development and maintenance, few studies have been conducted in pursuit of understanding logging usage in real-world software. Yuan et al., [2012] provides a quantitative characteristic study to investigate log message modifications on four open-source software systems by mining their revision history. Their study shows that developers spend a great effort to modify logging calls as after-thoughts, which indicates that they are not satisfied with the log quality in their first attempt. They also characterize where developers spend most of their time in modifying the log messages.

Yuan et. al. [2011] studies the problem of lack of log messages for error diagnosis and suggests to log when generic error conditions happens. LogEnhancer [Yuan et. al.] automatically enhances existing log message by detecting important variable values and inserting them into the log messages. However, these studies only consider code snippets containing bugs that are needed to be logged and do not consider other code snippets containing no bugs but still need to be logged. Moreover, these studies mainly research log message modifications and potential enhancements

of them, however, the focus of this study is on understanding where logging calls are used in the source code.

7.2 Correspondence

Several studies have been conducted to find similarities and differences between the source code fragments. Baxter et al. [1998] develop an algorithm to detect code clones in source code that uses hash functions to partition subtrees of ASTs of a program source code and then find common subtrees in the same partition through a tree comparison algorithm. Apiwattanapong et al. [2004] present a top-down approach to detect differences and correspondences between two versions of a Java program, through comparison of the control flow graphs created from the source code. Strathcona [Holmes et al., 2006] recommends relevant code snippet examples from a source code repository for the sake of helping developers to find examples of how to use an API by heuristically matching the structure of the code under development with the source code in the repository. Coogle [Sager et al., 2006] is developed to detect similar Java classes through converting ASTs to a normalized format and then comparing them through tree similarity algorithms. However, none of these approaches determines the detailed structural correspondences needed in our context.

Umami [Bradley et al., 2014] presents a new approach, called Matching via Structural generalization (MSG), to recommend replacements for API migration. He used the Jigsaw tool to find structural correspondences, however, their proposed algorithm does not suffice to our context since it does not construct a generalization to represent structural similarities and differences. It also does not take the required constraints in determining correspondences needed to solve our problem.

7.3 API usages patterns

Various data mining approaches has been used to extract API usages patterns out of the source code such as unordered pattern mining and sequential pattern mining [Robillard et al., 2013].

Unordered pattern mining, such as association rule mining and itemset mining, extracts a set of API usage rules without considering their order [Agrawal et al., 1994]. CodeWeb [Michail, 2000] uses data mining association rules to identify reuse patterns between a source code under development and a specific library. PR-Miner [Li and Zhou, 2005] uses frequent itemset mining to extract implicit programming rules from source code and detect violations. The sequential pattern mining technique is different from the unordered one in the way that it considers the order of API usage. As an example, MAPO [Xie and Pei, 2006] combines frequent subsequence mining with clustering to extract API usage patterns from the source code. The other technique for extracting API usage patterns is through statistical source code analysis. For example, PopCon [Holmes and Walker, 2007] is a tool developed to help developers understanding how to use APIs in their source code through calculating popularity statistics for each API of a library. Acharya et al. [2007] present a framework to extract API usage scenarios as partial orders. Specifications were extracted from frequent partial orders. They adapted a compile time model checker to generate control-flow-sensitive static traces of APIs, from which API usage scenarios were extracted. However, none of these approaches suffice to determine the detailed structural correspondences.

7.4 Antiunification

Antiunification is the problem of finding the most specific generalization of two terms. First-order syntactical antiunification was introduced by Plotkin [1970] and Reynolds [1970] independently. Burghardt and Heinz [1996] extend the notion of antiunification to E-antiunification to incorporate background knowledge to syntactical antiunification, which is required for some applications. antiunification has been applied in various studies for program analysis. Bulychev and Minea [2008] suggest an antiunification algorithm to detect clones in ASTs. Their approach consists of three stages: first, identifying similar statements through antiunification and classifying them into clusters; second, determining similar sequences of statements with the same Cluster identifier; third, refining candidate statement sequences using an antiunification based similarity measurement to

generate final clones. However, their approach does not construct a generalization by determining the structural correspondences. Cottrell et al. [2007] propose Breakaway to automatically determine structural correspondences between a pair of abstract syntax trees (ASTs) to create a generalized correspondence view. However, their approach does not allow us to detect the best structural correspondence for each node suited to our problem. Cottrell et al. [2008] develop Jigsaw to help developers integrate small-scale reused source code into their own code by determining structural correspondences through the application of higher-order antiunification modulo theories. However, considering the limitations of our study in determining correspondences, their approach does not suffice to construct a structural generalization needed in our context.

7.5 Clustering

Clustering is an unsupervised machine mining technique that aims to organize a collection of data into clusters, such that intra-cluster similarity is maximized and the inter-cluster similarity is minimized [Karypis, 1999] [Grira et al., 2004]. We divided existing clustering approaches into two major categories: partitional clustering and hierarchical clustering. Partitional clustering try to classify a data set into k clusters such that the partition optimizes a pre-determined criterion [Karypis]. The most popular partitional clustering algorithm is k-means, which repeatedly assigns each data point to a cluster with the nearest centroid and computes the new cluster centroids accordingly until a pre-determined number of clusters is obtained [Bouguettaya]. However, k-means clustering algorithm is not a good fit to our problem since it requires to predefine the number of clusters we want to come up with, which is not reasonable in our context.

Hierarchical clustering algorithms produce a nested grouping of clusters, with single point clusters at the bottom and an all-inclusive cluster at the top [Karypis, 1999]. Agglomerative hierarchical clustering is one of the main stream clustering methods [Day, 1984] and has applications in document retrieval [Voorhees, 1986] and information retrieval from a search engine query log [Beeferman et al., 2000]. It starts with singleton clusters, where each contains one data point. Then

it repeatedly merges the two most similar clusters to form a bigger one until a pre-determined number of clusters is obtained or the similarity between the closest clusters is below a pre-determined threshold value. Hierarchical clustering algorithms work implicitly or explicitly with the $n \times n$ similarity matrix such that an element in row i and column j represents the similarity between the i^{th} and the j^{th} clusters [Karypis, 1999].

There are various versions of agglomerative hierarchical algorithms that mainly differ in how they update the similarity between clusters. There are various methods to measure the similarity between clusters, such as single linkage, complete linkage, average linkage, and centroids [Rasmussen, 1992]. In the single linkage method, the similarity is measured by the similarity of the closest pair of data points of the two clusters. In the complete linkage method, the similarity is computed by the similarity of the farthest pair of data points of the two clusters. In the average linkage method, the similarity is measured by the average similarity of all pairwise similarities of data points of the two clusters. In the centroids methods, each cluster is represented by a centroid of all data points in the cluster, and the similarity between two clusters is measured by the similarity of the clusters' centroids. However, in our application, each cluster is composed of one AUAST, and the similarity between two clusters is measured by the similarity between the clusters' AUASTs, which is computed via antiunification.

7.6 Summary

Despite the great importance of logging and its various applications in software development and maintenance, few studies have focused on understanding logging usage in the source code. Some work has been done on characterizing log messages modifications made by developers and to help them enhance the content of log messages. However, to the best of our knowledge, no study has been conducted on characterizing where logging is used in the source code through determining structural correspondences. Several data mining and statistical source code analysis techniques have been used to extract API usage patterns, however, none of them enable us to determine the

detailed structural correspondences between source code fragments. On the other hand, using higher-order antiunification modulo theories and an agglomerative hierarchical clustering algorithm allow us to construct structural generalizations that describe the similarities and differences between logged Java classes and classifying logged Java classes into groups based on the structural correspondences, respectively.

Chapter 8

Conclusion

- Determining the detailed structural similarities and differences between source code fragments is a complex task
- It can be applied to solve several source code analysis problems, for example, characterizing logging practices
- logging is a pervasive practice and has various applications in software development and maintenance
- However, it is a challenging task for developers to understand how to use logging calls in the source code
- We have presented an approach to characterize where logging calls happen in the source code by means of structural generalization
- We have developed a prototype tool that:
  - detects potential structural correspondences using antiunification
  - uses several constraint to remove the correspondences that are not suited to our application
  - determines the best correspondences with the highest similarity
  - constructs the structural generalizations using antiunification
  - classifies the entities using a measure of similarity
- An experiment is conducted to evaluate our approach and tool
- Our experiment found that ...

- An experiment is conducted to characterize logging usage in three software systems
- In summary, our study makes the following contributions:
  -
  -

8.1 Future Work

- Future extensions could be applied to resolve the pitfalls of this study:
  - Data flow analysis techniques: to resolve the problem of inaccurate statement ordering
  - Further analysis: to detect and resolve all the conflicts happen in deciding the best correspondences
- To further validate our findings from the source code analysis:
  - a survey can be conducted to ask developers on the factors they consider when they want to decide on where to log
- Characterizing logging usage could be a huge step towards
  - improving logging practices by providing some guidelines that might help developers in making decisions about where to log.
  - developing recommendation support tools:
    * to save developers'time and effort
    * to improve the quality of logging practices

Bibliography

Peter Bulychev and Marius Minea. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*. Citeseer, 2009.

Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. The dimension of separating requirements concerns for the duration of the development lifecycle. In *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (at OOPSLA)*, 1999a.

Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. *ACM SIGPLAN Notices*, 34(10):325–339, 1999b.

Bradley Cossette, Robert Walker, and Rylan Cottrell. Using structural generalization to discover replacement functionality for api evolution. 2014.

Rylan Cottrell, Joseph JC Chang, Robert J Walker, and Jörg Denzinger. Determining detailed structural correspondence for generalization tasks. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 165–174. ACM, 2007.

Rylan Cottrell, Robert J Walker, and Jörg Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 214–225. ACM, 2008.

Rylan Cottrell, Brina Goyette, Reid Holmes, Robert J Walker, and Jorg Denzinger. Compare and contrast: Visual exploration of source code examples. In *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*, pages 29–32. IEEE, 2009.

Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, volume 9, pages 149–158, 2009.

Qiang Fu, Jian-Guang Lou, Qingwei Lin, Rui Ding, Dongmei Zhang, and Tao Xie. Contextual analysis of program logs for understanding system behaviors. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 397–400. IEEE Press, 2013.

Samudra Gupta. Pro apache log4j: Java application logging using the open source apache log4j api. *Apress®, USA*, 2005.

Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *USENIX Annual Technical Conference*, 2010.

Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 353–366, 2012.

Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH computer architecture news*, volume 38, pages 143–154. ACM, 2010.

Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 102–112. IEEE Press, 2012a.

Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):4, 2012b.