UNIVERSITY OF CALGARY

Automatically Characterizing Logging Usage: An Application of Anti-unification

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

November, 2016

Abstract

Logging has been a common practice to record the runtime behaviour of a software system, typically performed by inserting log statements in its source code. While several frameworks have been specifically created to help developers perform logging tasks, these do not provide guidance on where the log statements should be located in the source code. Thus, developers usually rely on their common sense to decide where to log. If logging is done properly, it can provide valuable information for software development and maintenance; if it is done poorly, system performance can degrade and maintenance can be made more difficult. Few studies have been conducted to characterize logging usage in real-world applications. This work tries to address the problem of where to log by proposing an automated approach that characterizes the location of log statements through the approximation of an anti-unification approach (specifically, higher-order anti-unification modulo theories) and a hierarchical clustering technique to construct a set of anti-unifiers, each describing the commonalities and differences between source code fragments that embody log statements. This approach has been reified in a prototype tool, called ELUS, that greedily identifies the best structural correspondences with respect to the highest similarity and some constraints. An empirical study was conducted by applying the tool on the source code of four open source systems and manually examining the generated anti-unifiers. The analysis resulted in five main categories of anti-unifiers in the logging usage. Two empirical evaluations were conducted in this work: (1) an experiment was conducted to evaluate the effectiveness of the proposed approach through the application of its supporting tool on a test suite; and (2) an experiment was performed to evaluate the quality of the anti-unifiers in describing the location of log statements in source code.

Acknowledgements

I would like to express my warmest gratitude to my supervisor, Dr. Robert J. Walker. I would like to thank him for his great support, guidance, and insightful discussions during my study. He always encouraged me to challenge myself by asking new research questions and to come up with new ways to solve a problem. I deeply appreciate his belief in me when I doubted myself, and his constant commitment and precious comments throughout the course of my research and thesis writing. I was always motivated by his support, understanding, and the freedom he gave me.

I am truly grateful to Dr. Jörg Denzinger, Dr. Günther Ruhe, and Dr. Christian Jacob for providing support and useful feedback for my work. Their technical knowledge and valuable feedback helped me a lot throughout the course of my research.

Many special thanks and love to my parents and my brothers for being such a great family. I am forever indebted to them for their support, patience, and encouragement throughout this work.

I would like thank my fellow labmates, Hamid, Elham, Soha, Mostafa, May, and Hao for their academic and friendship support. Their precious comments helped me a lot over the course of my research.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Abbreviations

AST	abstract syntax tree
AHC	agglomerative hierarchical clustering
AU	anti-unification
AUAST	anti-unifier abstract syntax tree
CAST	correspondence abstract syntax tree
HOAUMT	higher-order anti-unification modulo theories
JDT	Java Development Tools
LM	logged method
LUS	logged usage schema

Chapter 1

Introduction

Understanding the similarities and differences between a set of source code fragments is a potentially complex problem that has many actual or potential applications in various software engineering research areas, such as: code clones detection [Bulychev and Minea, 2009]; automating source code reuse [Cottrell et al., 2008]; recommending application programming interface (API) replacements amongst various versions of a software library [Cossette et al., 2014]; collating API usage patterns; and automating the merge operation in a version control system. As a specific application, the focus of this study is on characterizing where log statements are used in source code via the determination of structural correspondences between a set of source code fragments enclosing them.

Logging is a conventional programming practice that has usually been used by developers to diagnose the presence or absence of a particular event in a system, to understand the state of an application, and to follow a program's execution flow to find the root causes of an error. The importance of logging is notable in its various applications during software development, such as problem diagnosis [Lou et al., 2010], system behavioural 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]. Despite the significance of logging for software development and maintenance, few studies have been conducted on understanding its usage in real-world applications, as it has been considered to be a trivial task [Clarke et al., 1999a,b]. However, the availability of several complex frameworks (e.g., Apache log4j, SLF4J) that assist developers in logging suggests that in practice effective logging is not a straightforward task. In addition, a study by Yuan et al. [2012b] showed that developers expend great effort in modifying their logging practices as an afterthought. This indicates that it is not that simple for developers to perform logging effectively on their first attempt.

The challenges associated with high quality logging arises form the fact that developers are usually left with the burden of deciding where and what to log manually, thus log statements can be inserted in various locations of source code. For example, a developer may decide to insert log statements at the start and end of every method to record the occurrence of every event of an application. However, three main problems are associated with excessive logging. First, it can produce a lot of redundant information that makes the system log analysis confusing and misleading. Second, excessive logging is costly. It requires extra time and effort to write, debug, and maintain the logging code. Third, it can generate system resource overhead and thus the application performance will be negatively affected. On the other hand, insufficient usage of log statements may result in the loss of run-time information necessary for software analysis. Therefore, logging should be done in an appropriate manner to be effective.

Research on the problem of understanding logging practices can be divided into two main topics: the context and the location of log statements. The context refers to the log text messages, while the location refers to where logging statements are used in source code. The context of log statements is important to perform high quality logging, as it provides necessary information needed for system analysis. The location of log statements also 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 within a system. A few studies have been conducted on characterizing log text message modifications [Yuan et al., 2012b] and developing tools to automatically enhance the context of existing logging statements [Yuan et al., 2012c, 2010]. Yuan et al. [2012a] proposed Errlog to automatically insert additional log statements into a software system to log all the generic exceptions in order to enhance failure diagnosis. Zhu et al. [2015] applied machine learning techniques to determine the important factors impacting the location of the log statements in source code. In this study, I address the problem of understanding where to log by developing an automated approach that investigates the feasibility of finding patterns of where log statements occur in source code through the construction of a detailed view of structural generalizations representing the commonalities and differences between source code fragments that contain logging statements.

1.1 Programmatic support for logging

A typical log statement takes parameters including a log text message and a verbosity level. A log text message consists of static text that describes 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 log statements from the Apache 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 causes of an error or in understanding the application behaviour. The debug level provides useful information for debugging an application, and it is usually used by developers only during the development phase. In general, verbosity level is used for classification, in order 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: Log statement examples from the Apache log4j framework.

1.2 Broad thesis overview

I aim to create an approach that provides a description of where logging statements are used in source code by constructing generalizations that represent the structural similarities and differences between methods that make use of log statements, which I call *logged methods* (LMs). In order to evaluate this idea, I implemented the approach to operate on programs written in the Java programming language. To determine how to construct generalizations using the syntax and semantics of the Java programming language, I looked to previous research conducted by Cottrell et al. [2008] that determined the structural correspondences between two Java source code fragments through the application of approximated anti-unification, such that one fragment can be integrated with the other one for small-scale code reuse. However, my problem context is different, as I need to generalize a set of source code fragments with special attention to log statements. Therefore, my approach must take the logs into account when I perform the generalization task via the determination of structural correspondences.

My approach to characterizing logging usage proceeds in four steps (as shown in Figure 1.2). First, potential structural correspondences are determined between the abstract syntax trees (ASTs) of LMs in a pairwise manner, and stored in a novel structure: the *anti-unifier AST* (AUAST), which allows the application of anti-unification on AST structures. Second, I use an approximated anti-unification algorithm to construct a structural generalization (an anti-unifier) representing the commonalities and differences between AUAST pairs, which employs a greedy selection algorithm to approximate the best anti-unifier for the problem by determining the most similar correspondence for each node. The anti-unification algorithm also applies some constraints prior to determining the best correspondences, in order to prevent the anti-unification of log statements with any other types of nodes in the tree structure. The anti-unifier is constructed through the anti-unification of each AUAST node with its best correspondence and then a measure of structural similarity is developed between the two AUASTs. In the third step, I employ a hierarchical clustering algorithm to group the AUASTs into a number of clusters using the structural similarity measure and I then create a structural generalization from each cluster. The last step involves creating a detailed view of each structural generalization, which I called *logging usage schema* (LUS), that represent the structural commonalities and differences between the set of LMs within each cluster. I manually went through the LUSs to characterize the location of logging statements in source code.

To evaluate the approach, I implemented it in a tool called ELUS, written in the Java programming language. I used the Eclipse JDT framework to extract the AST of LMs from a Java program, and employed the Jigsaw framework developed by Cottrell et al. [2008] to find potential structural correspondences. My anti-unifier building tool (built atop Jigsaw) is applied to construct the structural generalizations, and my clustering tool is developed atop of it to perform the clustering algorithm .

To characterize logging usage using my approach, I applied ELUS to the source code of four open-source software systems: Tomcat, Hibernate, Camel, and Solr. My analysis has resulted in five main categories of anti-unifiers in the logging usage. To evaluate the usefulness of my findings, I have conducted an empirical study to asses the performance of ELUS. This experiment shows that ELUS has an average precision of 84% and an average recall of 80%, and thus can be used to automatically construct the anti-unifiers of logging usage in source code.

.

1.3 Thesis statement

The thesis of this work is to characterize where log statements occur in source code by constructing structural generalizations that describe the commonalities and differences between source code fragments containing log statements, thus providing the developers with some guidelines on where to use them effectively in source code.

1.4 Thesis organization

The remainder of the thesis is organized as follows.

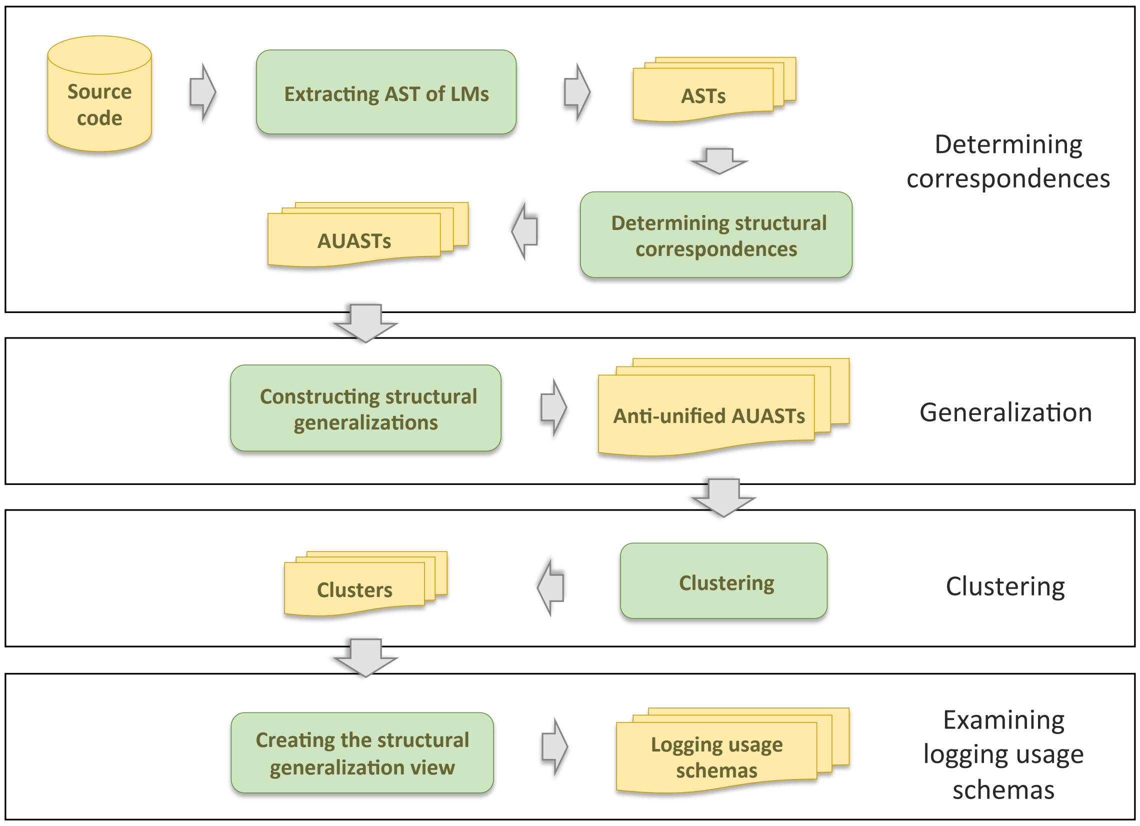

Source code
Extracting AST of LMs
ASTs
Determining correspondences
AUASTs
Determining structural correspondences
Constructing structural generalizations
Anti-unified AUASTs
Generalization
Clusters
Clustering
Clustering
Creating the structural generalization view
Logging usage schemas
Examining logging usage schemas

Figure 1.2: Overview of the approach.

Chapter 2 motivates the problem of understanding where to use log statements in 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 not sufficiently supported.

Chapter 3 provides background information that I build atop: abstract syntax trees (ASTs), which are the basic structure I will use for describing software source code; the Eclipse JDT, an industrial framework for producing and manipulating ASTs for source code written in the Java programming language; anti-unification, which is a theoretical approach for constructing structural generalizations; and on Jigsaw, a research tool based on the Eclipse JDT for performing anti-unification.

Chapters 4, 5, and 6 present the first three steps of my approach. Determining structural correspondences between AUASTs; constructing structural generalizations from an AUAST pair; and classifying a set of AUASTs into separate clusters, respectively. In each chapter, I discuss the implementation of my approach as an Eclipse plug-in, and conduct an experimental study to assess the effectiveness of my approach by applying its tool support on a sample test suite extracted from a real software system.

Chapter 7 presents an empirical study I conducted to characterize the location of log statements of four open-source software systems. Chapter 8 discusses the results and findings of my work, threats to its validity, and remaining issues. Chapter 9 describes work related to my research problem and how it does not adequately address the problem. Chapter 10 concludes the dissertation and presents the contributions of this study and future work.

Chapter 2

Motivational Scenario

Printing messages to the console or to a log file is an integral part of software development and can be used to test, debug, and understand what is happening inside an application. In the 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 in the logging quality suggests that using print statements is not sufficient to perform logging practices in real-world applications.

The logging frameworks offer many more facilities that are not provided by the print statements. For example, most logging frameworks (e.g., log4j, SLF4J, java.util.logging) use different verbosity levels to control the types of information needed to be logged. That is, by logging at a particular verbosity level, logs at that level and higher levels will be recorded whereas the logs at lower levels will be discarded. Each of these verbosity levels can be used for different applications during software development. For example, the debug log level messages can be used in a test environment, while the error log level messages can be used in a production environment. This feature not only produces fewer log messages for each level, but also improves the performance of an application. Also, most logging frameworks allow the production of formatted log messages, which makes it easier for a developer to monitor the behavior of a system. In addition, when a developer is working on a server side application, the only way to know what is happening inside the server is by monitoring the log files. Although logging is a valuable 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. As latency and speed are major concerns for most software systems, it is necessary for a developer to understand and learn logging practices in great detail in order to perform them in an efficient manner.

To illustrate the inherent challenges of effectively performing logging practices in software systems, one may consider a scenario in which a developer is asked to log an event-based mechanism of a text editor tool written in the Java programming language. In this scenario, the developer is trying to log a Java class of the system (Figure 2.1) using the Apache log4j framework. She knows that components of this application register with the EditBus class to receive messages that reflect changes in the application's state, and that 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 subscribe to the EditBus and implement the EBComponent interface define the method EBComponent.handleMessage(EBMessage) to handle a message sent on by the EditBus. To perform this logging task, the developer might ask herself several fundamental questions, mostly related to where and what to log.

Her first solution might be 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, and will produce redundant information. She assumes that the more she logs, the more she performs file I/O, which slows down the application. Therefore, she decides to log only important information necessary to debug or troubleshoot potential problems. She proceeds to identify the information needed to be logged and then decides on where to use log statements. 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 find the root causes of potential problems in sending messages. She simply wants to begin by using a log statement at the start of the send() method (line 2 of Figure 2.2) to log the information. However, she realizes that this log statement does not allow her to log all the information she wants, as the time variable is not initialized at the beginning of this method. Therefore, she proceeds to examine the body of the send() method line-by-line and uses another log statement after the time variable is initialized. She aims to log the transmission time in case of potential problems in sending messages. Therefore, she decides to insert the logging call inside

```java
public class EditBus {
    private static ArrayList components = new ArrayList();
    private static EBComponent[] copyComponents;

    private EditBus() {
    }

    public static void addToBus(EBComponent comp) {
        synchronized(components) {
            components.add(comp);
            copyComponents = null;
        }
    }

    public static void removeFromBus(EBComponent comp) {
        synchronized(components) {
            components.remove(comp);
            copyComponents = null;
        }
    }

    public static EBComponent[] getComponents() {
        synchronized(components) {
            if (copyComponents == null) {
                EBComponent[] arr = new EBComponent[components.size()];
                copyComponents =
                    (EBComponent[])components.toArray(arr);
            }
        }
        return copyComponents;
    }

    public static void send(EBMessage message) {
        EBComponent[] comps = getComponents();
        for(int i = 0; i < comps.length; i++) {
            EBComponent comp = comps[i];
            long start = System.currentTimeMillis();
            comp.handleMessage(message);
            long time = (System.currentTimeMillis() - start);
        }
    }
}
```

Figure 2.1: The EditBus class.

an **if** statement that logs the value of the variable time, if it is not within a valid range (shown in lines 9–11 of Figure 2.3).

She also believes that it is important to log an error if any problems occur 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 log statement 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 will not allow her to reach the desired functionality, as it does not reveal to which component the problem is related. Thus, she decides to relocate 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).

Figure 2.6 shows the developer's final determination of the usage of log statements to perform the logging task of the EditBus class. By making appropriate decisions about where to use log statements, the developer is in a good position to proceed to write the logging messages by examining the remaining conceptually complex questions. What specific information should I log? How should I choose the log message format? Which information goes to which level of logging? If the developer had reached this point more easily and quickly, she would have had more time and energy to make decisions about the remaining issues and could have completed the logging practice in a timely and appropriate manner.

2.1 Summary

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

```java
public static void send(EBMessage message){
    //log statement
    EBComponent[] comps = getComponents();
    for (int i = 0; i < comps.length; i++) {
        EBComponent comp = comps[i];
        long start = System.currentTimeMillis();
        comp.handleMessage(message);
        long time = (System.currentTimeMillis() - start);
    }
}
```

Figure 2.2: The developer's initial determination of the usage of log statements for the send(EBMessage) method.

```java
public static void send(EBMessage message) {
    //log statement
    EBComponent[] comps = getComponents();
    for(int i = 0; i < comps.length; i++) {
        EBComponent comp = comps[i];
        long start = System.currentTimeMillis();
        comp.handleMessage(message);
        long time = (System.currentTimeMillis() - start);
        if (time >= 1000000) {
            //log statement
        }
    }
}
```

Figure 2.3: The developer's second determination of the usage of log statements for the send(EBMessage) method.

```
public static void send(EBMessage message){
    try {
        //log statement
        EBComponent[] comps = getComponents();
        for(int i = 0; i < comps.length; i++) {
            EBComponent comp = comps[i];
            long start = System.currentTimeMillis();
            comp.handleMessage(message);
            long time = (System.currentTimeMillis() - start);
            if (time >= 1000000) {
                //log statement
            }
        }
    } catch(Throwable t) {
        //log statement
    }
}
```

Figure 2.4: The developer's third determination of the usage of log statements for the send(EBMessage) method.

```
public static void send(EBMessage message) {
    //log statement
    EBComponent[] comps = getComponents();
    for (int i = 0; i < comps.length; i++) {
        try {
            EBComponent comp = comps[i];
            long start = System.currentTimeMillis();
            comp.handleMessage(message);
            long time = (System.currentTimeMillis() - start);
            if (time >= 1000000) {
                //log statement
            }
        } catch(Throwable t) {
            //log statement
        }
    }
}
```

Figure 2.5: The developer's fourth determination of the usage of log statements for the send(EBMessage) method.

```
public class EditBus {
    private static ArrayList components = new ArrayList();
    private static EBComponent[] copyComponents;

    private EditBus() {
    }

    public static void addToBus(EBComponent comp) {
        synchronized(components) {
            components.add(comp);
            copyComponents = null;
        }
    }

    public static void removeFromBus(EBComponent comp) {
        synchronized(components) {
            components.remove(comp);
            copyComponents = null;
        }
    }

    public static EBComponent[] getComponents() {
        synchronized(components) {
            if (copyComponents == null) {
                EBComponent[] arr = new EBComponent[components.size()];
                copyComponents = (EBComponent[])components.toArray(arr);
            }
        }
        return copyComponents;
    }

    public static void send(EBMessage message) {
        //log statement
        EBComponent[] comps = getComponents();
        for(int i = 0; i < comps.length; i++) {
            try {
                EBComponent comp = comps[i];
                long start = System.currentTimeMillis();
                comp.handleMessage(message);
                long time = (System.currentTimeMillis() - start);
                if (time >= 1000000) {
                    //log statement
                }
            } catch(Throwable t) {
                //log statement
            }
        }
    }
}
```

Figure 2.6: The developer's final determination of the usage of log statements for the EditBus class.

Chapter 3

Background

A programming language is described by the combination of its syntax and semantics. The syntax concerns the legal structures of programs written in the programming language, while the semantics is about the meaning of every construct in that language. Furthermore, the abstract syntactic structure of source code written in a programming language can be represented as an *abstract syntax tree* (AST), in which nodes are occurrences of syntactic structures and edges represent nesting relationships. Since ASTs will be the form in which I represent and analyze source code, I need a means to generalize sets of ASTs in order to understand their commonalities while abstracting away their differences. The theoretical framework of anti-unification is presented as that means.

In this chapter, ASTs are described in Section 3.1, along with their more concrete counterparts, concrete syntax trees. A specific, industrial framework for creating and manipulating ASTs for source code written in the Java programming language—the Eclipse JDT—is described in Section 3.2. Anti-unification is summarized in Section 3.3, starting with its most basic form, first-order anti-unification, and progressing to the form that I will make use of, higher-order anti-unification modulo equational theories, in Section 3.4. A research approach, built atop the Eclipse JDT, for performing anti-unification on Java ASTs—the Jigsaw framework—is described in Section 3.6. In the Section 3.7, I provide some background information about clustering, existing clustering techniques, and the agglomerative hierarchical clustering algorithm, as a technique I used to cluster logged methods into separate groups based on a similarity measurement.

3.1 Concrete syntax trees and abstract syntax trees

A concrete syntax tree is a tree $T = (V, E)$ whose vertices V (equivalently, nodes) represent the syntactic structures (equivalently, syntactic elements) of a specific program written in a specific

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Figure 3.1: A simple example Java program.

programming language and whose directed edges E represent the nesting relationships amongst those syntactic structures. Non-leaf nodes in a concrete syntax tree (also called a parse tree) represent the grammar productions that were satisfied in parsing the program it represents; leaf nodes represent the concrete lexemes, such as literals and keywords.

I focus on the Java programming language and I make use of the grammar in the language specification [Gosling et al., 2012, Chapter 18] to determine the form of the concrete syntax trees. Non-leaf node names are represented by names in "camel-case" written in italics. Consider the trivial program in Figure 3.1; its concrete syntax tree is represented in Figure 3.2.

Beyond the fact that the concrete syntax tree is rather verbose and thus occupies a lot of space even for a trivial example, I can see two key problems with it: (1) there are a multitude of redundant nodes such as *expression1*, *expression2*, and *expression3* that are present solely for purposes of creating an unambiguous grammar; and (2) there are no nodes that express key concepts, such as "method declaration" and "method invocation", that should be obviously present in the example program.

To address these problems, concrete syntax trees are converted to abstract syntax trees (ASTs). An AST is similar in concept to a concrete syntax tree but it does not generally represent the parsing steps followed to differentiate different kinds of syntactic structure. The node types are chosen to represent syntactical concepts; I use the grammar presented for exposition by Gosling et al. [2012], which differs markedly from the grammar they propose in their Chapter 18 for efficient parsing. Note that a given node type constrains the kinds and numbers of child nodes that it possesses. The AST derived from the concrete syntax tree of Figure 3.2 is shown in Figure 3.3. Note that,

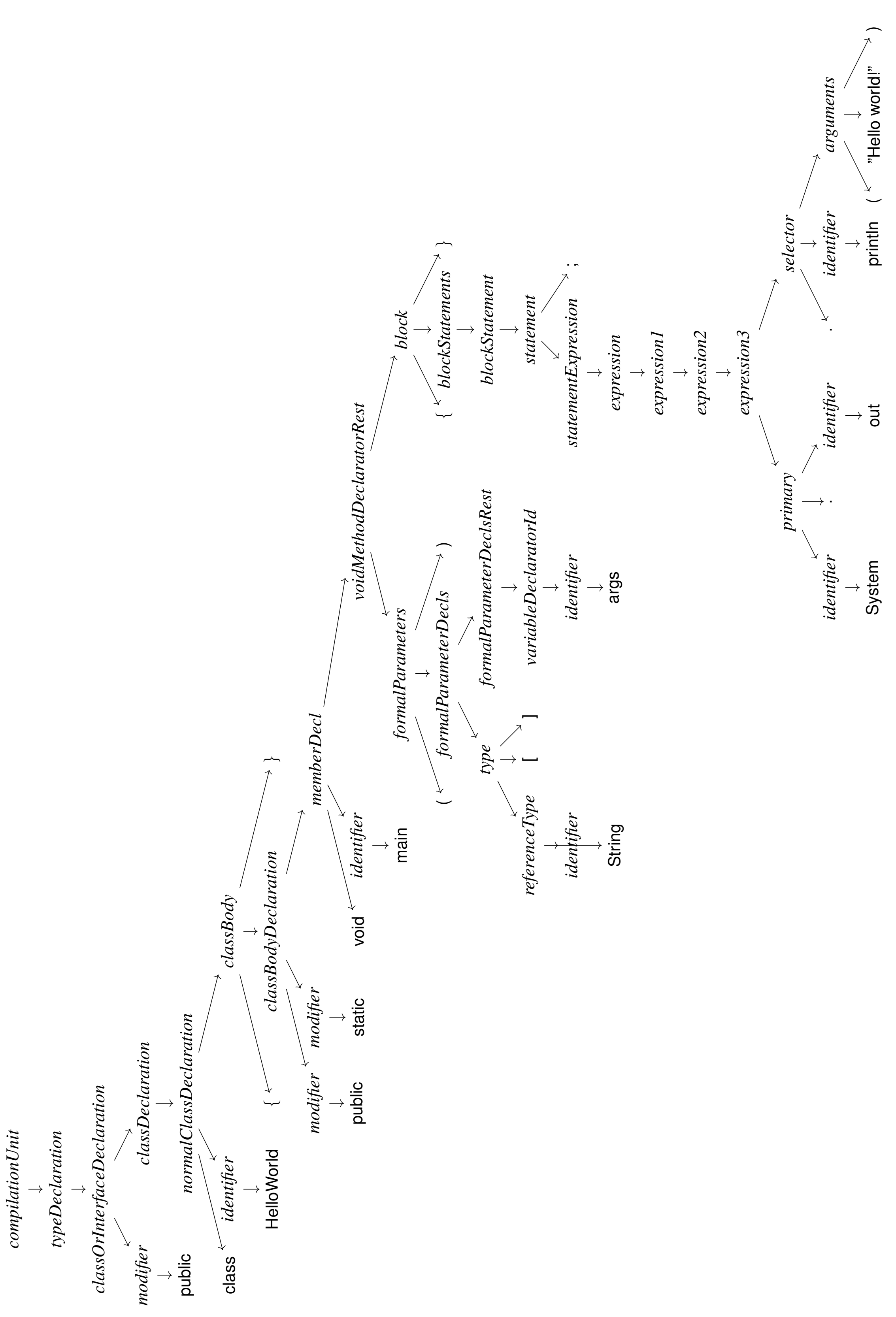


Figure 3.2: The concrete syntax tree for the program of Figure 3.1.

although I know that (for any normal program) System refers to the class java.lang.System and out is a static field on that class, non-normal programs can occur and a pure syntactic analysis cannot rule out that System is a package and that out is a class therein declaring a static method println(String).

This is still verbose, so in practice we elide details that are implied or otherwise trivial, to arrive at a more abstract AST as shown in Figure 3.4.

3.2 Eclipse JDT

The Eclipse Java Development Tools (JDT) framework provides APIs to access and manipulate Java source code via ASTs. An AST represents Java source code in a tree form, where the typed nodes represent instances of certain syntactic structures from the Java programming language. Each node type (in general) takes a set of child nodes, also typed and with certain constraints on their properties. Groups of children are named on the basis of the conceptual purpose of those groups; optional groups can be empty, which we can represent with the NIL element. For example, the simple AST structure of two sample LMs in Figures 3.5 an 3.6 is shown in Figure 3.7, with the log statements highlighted in yellow.

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

- *Simple structural properties:* These 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). For example, all the *identifier* nodes in Figure 3.3 fall in this case; each references an instance of String representing the string that constitutes the identifier.
- *Child structural properties:* These involve situations where the value is a single AST node (e.g., name property of a method declaration node). For example, the *classDec-*

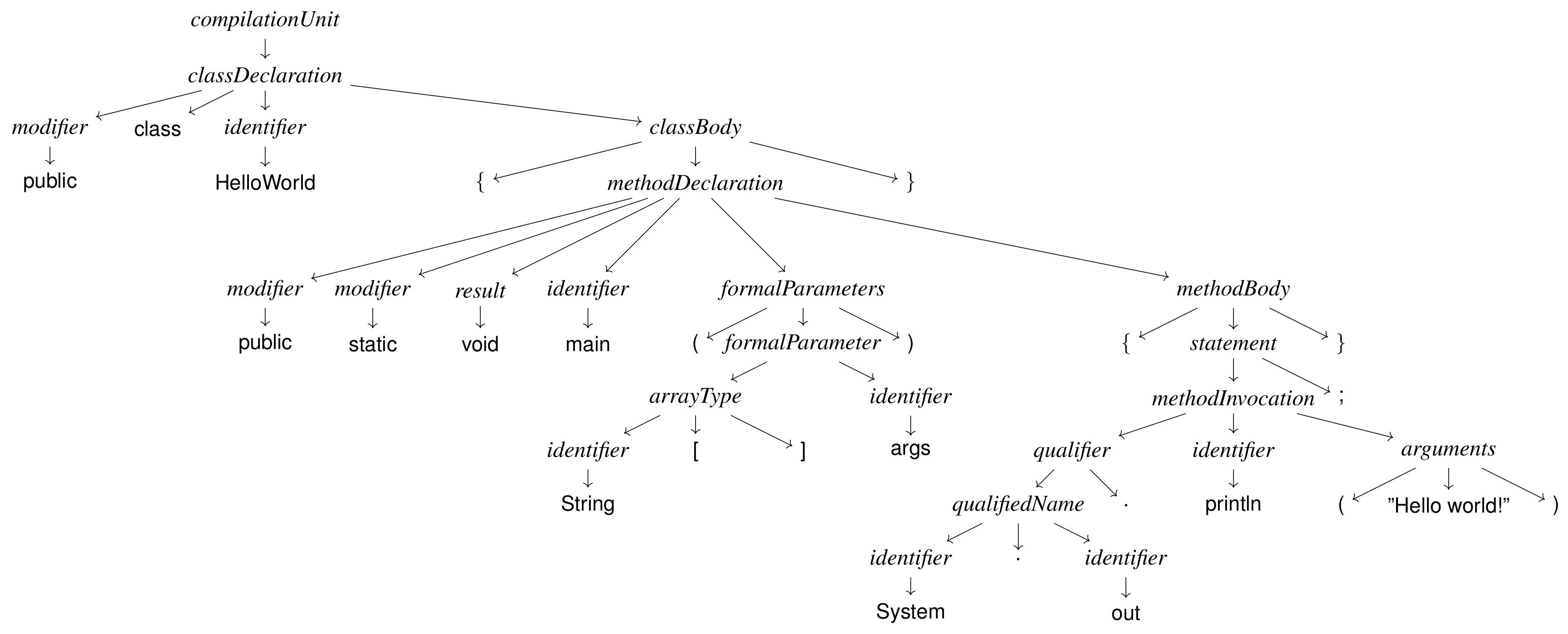


Figure 3.3: The abstract syntax tree derived from the concrete syntax tree of Figure 3.2.

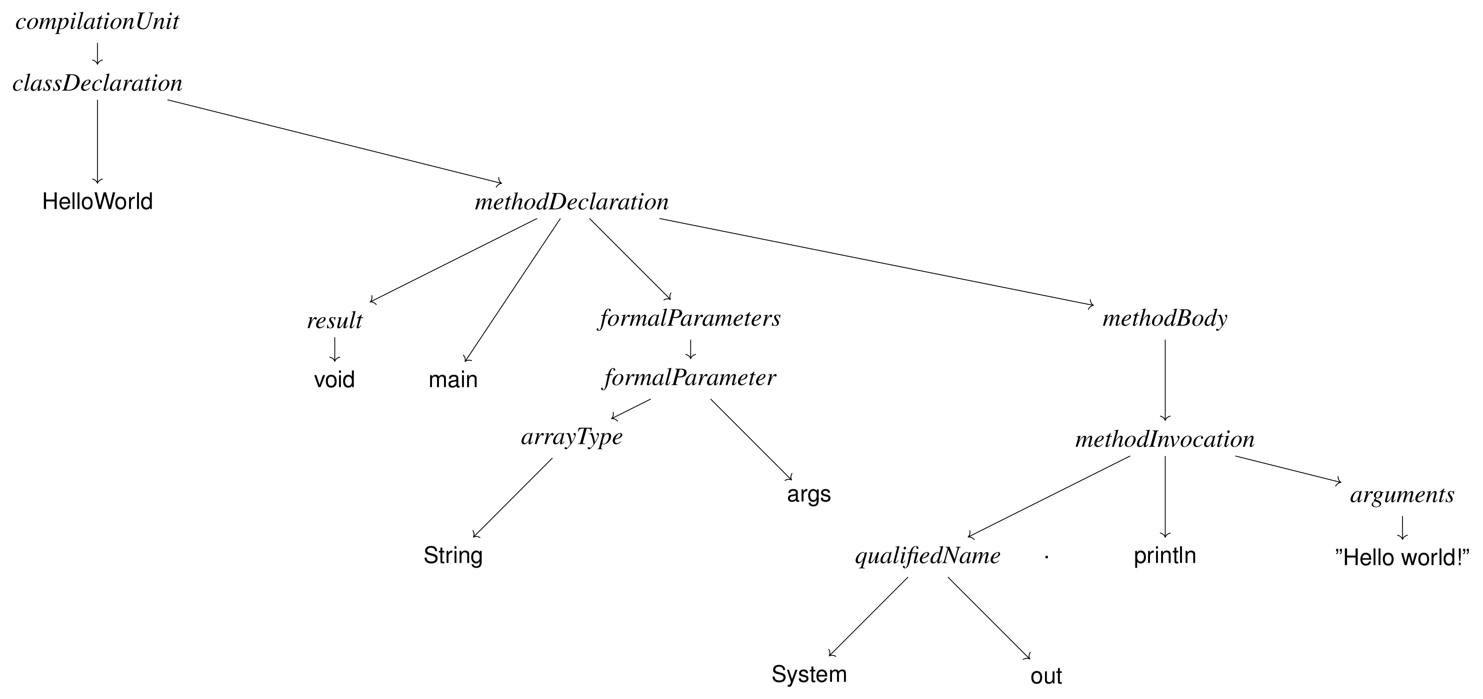


Figure 3.4: A more abstract AST derived from the concrete syntax tree of Figure 3.2.

```
public void handleMessage(EBMessage message) {
   if (seenWarning)
      return;
   seenWarning = true;
   Log.log(Log.WARNING, this, getClassName() + " should extend EditPlugin not EBPlugin
        since it has an empty " + handleMessage());
}
```

Figure 3.5: A Java method that uses a log statement. This will be referred to as Example 1.

```
public void actionPerformed(ActionEvent evt) {
   EditAction action = context.getAction(actionName);
   if (action == null) {
      Log.log(Log.ERROR, this, "Unknown action: " + actionName);
   }
   else{
      context.invokeAction(evt, action);
   }
}
```

Figure 3.6: A Java method that uses a log statement. This will be referred to as Example 2.

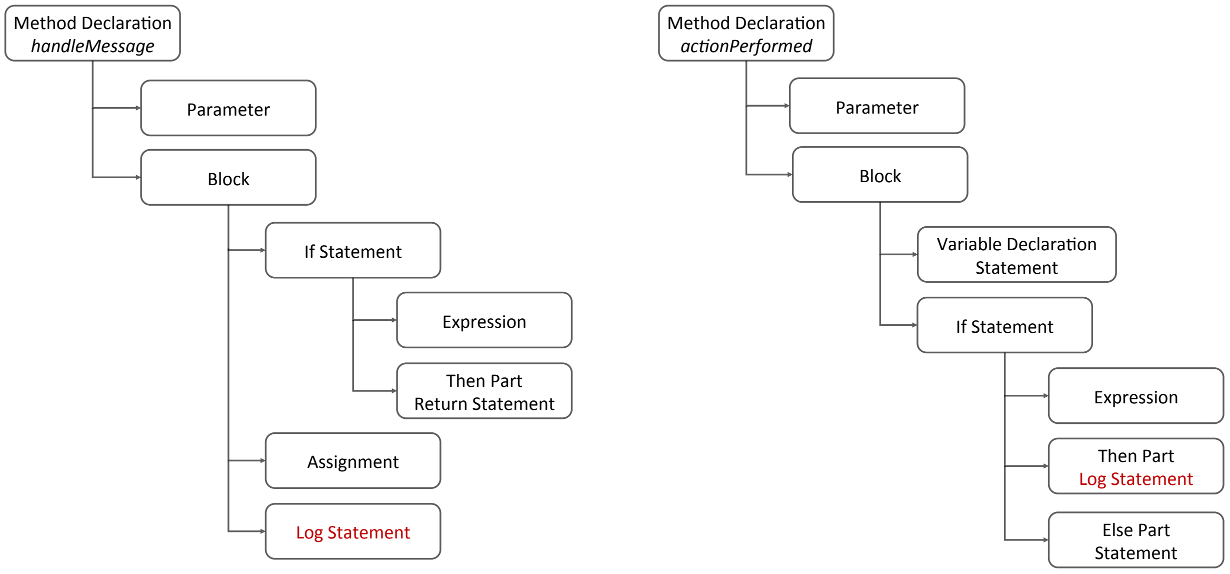


Figure 3.7: Simple AST structure of the examples in Figures 3.5 and 3.6.

laration node in Figure 3.3 has a single child that represents its name as an *identifier* node.

- *Child list structural properties*: These involve situations where the value is a list of child nodes. For example, the *classDeclaration* node in Figure 3.3 can possess multiple *modifier*s.

As an example, the ASTs of the log statements at line 4 of Figure 3.5 and Figure 3.6 can be represented respectively as:

- *methodInvocation*(
  qualifiedName(Log, *identifier*(log)),
  arguments(
  qualifiedName(Log, *identifier*(WARNING)),
  thisExpression(),
  additionExpression(
  methodInvocation(*identifier*(getClassName), *arguments*()),
  stringLiteral(" should extend EditPlugin not EBPlugin since it has an empty "),
  methodInvocation(*identifier*(handleMessage), *arguments*())))))

- *methodInvocation*(
  qualifiedName(Log, *identifier*(log)),
  arguments(
  qualifiedName(Log, *identifier*(ERROR)),
  thisExpression(),
  additionExpression(
  stringLiteral("Unknown action: "),
  identifier(actionName))))

3.3 First-order anti-unification

This section defines terms, substitutions, applying a substitution to a term, and instances of a term, as the requirements needed to describe first-order anti-unification.

Definition 3.3.1 (First-Order Term)**.** Given a set V of (structural) variable symbols, a set C of constant symbols, and sets F_n of n-ary function symbols for all $n \in \mathbb{N}$, the set T of *first-order terms* is defined as the smallest set satisfying the recursion: (1) $V \subseteq T$; (2) $C \subseteq T$; and (3) for all n first-order terms $t_1, \ldots, t_n$ and n-ary function symbol $f \in F_n$, $f(t_1, \ldots, t_n) \in T$.

Constant symbols can equivalently be considered 0-ary function symbols, with the appropriate adjustments to the above definition. For notational convenience, I use identifiers starting with a lowercase letter to represent function symbols (e.g., $f(a, b)$, $g(a, b)$) and constants (e.g., a, b), while variables are represented by identifiers starting with an uppercase letter (e.g., X, Y). The following are examples of a first-order term:

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

Note that for any first-order term there is a unique, equivalent tree and vice versa: constant symbols and variable symbols are leaf nodes, while function symbols are non-leaf nodes; a function with given arguments is represented by a non-leaf node (representing the function symbol) with directed edges pointing to leaf nodes representing each argument. For example:

- Y
- a

- $$\begin{tikzcd} & f \arrow[dl] \arrow[dr] & \\ X & & c \end{tikzcd}$$

- $$\begin{tikzcd} & & & f \arrow[dll] \arrow[d] \arrow[drr] & & & \\ & g \arrow[dl] \arrow[dr] & & Y & & g \arrow[dl] \arrow[dr] & \\ X & & b & & a & & Z \end{tikzcd}$$

Definition 3.3.2 (First-Order Substitution)**.** A first-order substitution σ is a mapping from variables V to first-order terms T: $\sigma : V \mapsto T$. The notation $\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ is used to express a substitution of each of a set of variables v_i by a corresponding first-order term t_i.

Definition 3.3.3 (Applying a First-Order Substitution)**.** Applying a first-order substitution $\sigma = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ to a first-order term t results in the simultaneous replacement of all occurrences in t of each variable v_i by its corresponding first-order term t_i as defined by the first-order substitution. This is denoted with the expression $\sigma(t)$.

As an example, applying the first-order substitution $\sigma = \{X \mapsto a, Y \mapsto b\}$ to the first-order term $f(X, Y)$ results in the replacement of all occurrences of the variable X by the first-order term a and all occurrences of the variable Y by the first-order term b, and thus $\sigma(f(X, Y)) = f(a, b)$.

Definition 3.3.4 ((First-Order) Instance)**.** For first-order terms t_1 and t_2, t_2 is called an *instance* of t_1 if there exists a first-order substitution σ such that $\sigma(t_1) = t_2$.

Definition 3.3.5 (First-Order Anti-unifier (and Unifier))**.** A *first-order unifier* of two terms t_1 and t_2 is a substitution σ such that $\sigma'(t_1) = \sigma'(t_2)$. The term t is a *first-order anti-unifier* (or generalization) for first-order terms t_1 and t_2, if and only if there exist first-order substitutions σ_1 and σ_2 such that $\sigma_1(t) = t_1$ and $\sigma_2(t) = t_2$.

Note that this terminology (and the corresponding higher-order terminology) is not symmetrical between unification and anti-unification. Fortunately, I do not need to make further use of

$$\begin{tikzcd}
 & \begin{array}{c}\boxed{f(X,Y)}\\ \text{anti-unifier}\end{array} \arrow[dl, "\sigma_1 = Y \mapsto b"'] \arrow[dr, "\sigma_2 = X \mapsto a"] & \\
\boxed{f(X,b)} \arrow[dr, "\sigma' = \{X \mapsto a, Y \mapsto b\}"'] & & \boxed{f(a,Y)} \arrow[dl, "\sigma' = \{X \mapsto a, Y \mapsto b\}"] \\
 & \begin{array}{c}\boxed{f(a,b)}\\ \text{unified term}\end{array} &
\end{tikzcd}$$

Figure 3.8: Unification and anti-unification of the terms $f(X, b)$ and $f(a, Y)$.

unification beyond this chapter, so this point of confusion remains as a mere curiosity from a practical perspective.

A useful, first-order anti-unifier will contain only common pieces of the original terms, while the differences are abstracted away by replacing them with variable symbols. An anti-unifier for a pair of terms always exists since we can anti-unify any two terms by the term $X \in V$, i.e., a single variable. However, anti-unification usually aims to find the *most specific anti-unifier* (MSA), that is, t is the MSA of two first-order terms t_1 and t_2 if and only if there exists no anti-unifier t' for t_1 and t_2 such that $\sigma(t) = t'$ for some first-order substitution σ.

As an example, an anti-unifier of the first-order terms $f(X, b)$ and $f(a, Y)$ is the first-order term $f(X, Y)$, containing common pieces of the two original, first-order terms. The variable Y in the anti-unifier $f(X, Y)$ can be substituted by the first-order term b to re-create $f(X, b)$ (with $\sigma_1 = \{Y \mapsto b\}$) and the variable X in the anti-unifier can be substituted by the first-order term a to re-create $f(a, Y)$ (with $\sigma_2 = \{X \mapsto a\}$), as depicted in Figure 3.8.

The MSA should preserve as much of the common pieces of both original first-order terms as possible; however, first-order anti-unification fails to capture complex commonalities, as first-order substitutions only replace first-order variables by first-order terms. That is, when two first-order terms differ in function symbols, first-order anti-unification fails to retain common details of the

$$\begin{tikzcd} & \boxed{X} \arrow[dl, "\sigma_1 = X \mapsto f(a,b)"'] \arrow[dr, "\sigma_2 = X \mapsto g(a,b)"] & \\ \boxed{f(a,b)} & & \boxed{g(a,b)} \end{tikzcd}$$

Figure 3.9: First-order anti-unification of the terms $f(a, b)$ and $g(a, b)$.

arguments in both terms. For example, the first-order anti-unifier of the terms $f(a, b)$ and $g(a, b)$ is X as depicted in Figure 3.9.

3.3.1 Higher-order anti-unification

Higher-order anti-unification generalizes first-order anti-unification to permit function symbols to be substituted for certain variable symbols (functional ones, to be precise). The needed formal definitions follow.

Definition 3.3.6 (Higher-Order Term)**.** Given a set V of variable symbols, a set C of constant symbols, sets F_n of n-ary function symbols for all $n \in \mathbb{N}$, and sets $\mathbf{V}_m$ of m-ary functional variable symbols, the set $\hat{T}$ of *higher-order terms* is defined as the smallest set satisfying the recursion: (1) $V \subseteq \hat{T}$; (2) $C \subseteq \hat{T}$; (3) for all n higher-order terms $\hat{t}_1, \ldots, \hat{t}_n$ and n-ary function symbol $f \in F_n$, $f(\hat{t}_1, \ldots, \hat{t}_n) \in \hat{T}$; and (4) for all m higher-order terms $\hat{t}_1, \ldots, \hat{t}_m$ and m-ary functional variable symbol $F \in \mathbf{V}_m$, $F(\hat{t}_1, \ldots, \hat{t}_n) \in \hat{T}$.

Note that any first-order term on V, C, and F_n will also be (a degenerate case of) a higher-order term on V, C, F_n, and $\mathbf{V}_m$ for all $\mathbf{V}_m$.

Definition 3.3.7 (Higher-Order Substitution)**.** A higher-order substitution $\hat{\sigma}$ is a mapping from variables V to higher-order terms $\hat{T}$ and, for all $m \in \mathbb{N}$, from m-ary functional variables $\mathbf{V}_m$ to m-ary functional symbols F_m; in other words, $\sigma : (V \mapsto T) \cup (\forall m \in \mathbb{N}, \mathbf{V}_m \mapsto F_m)$. The notation $\{\hat{v}_1 \mapsto \hat{t}_1, \ldots, \hat{v}_n \mapsto \hat{t}_n\}$ is used to express a substitution of each of a set of variables and functional variables $\hat{v}_i$ by a corresponding higher-order term $\hat{t}_i$ or functional symbols, where it is

$$\begin{tikzcd}
 & \boxed{F(a,b)} \arrow[dl, "\hat{\sigma}_1 = \{F \mapsto f\}"'] \arrow[dr, "\hat{\sigma}_2 = \{F \mapsto g\}"] & \\
\boxed{f(a,b)} & & \boxed{g(a,b)}
\end{tikzcd}$$

Figure 3.10: A higher-order anti-unification of the terms $f(a,b)$ and $g(a,b)$.

to be understood that a m-ary functional variable may only be substituted by a m-ary functional symbol and that a variable may only be substituted by a higher-order term.

Note that a first-order substitution constitutes (a degenerate case of) a higher-order substitution.

Definition 3.3.8 (Applying a Higher-Order Substitution)**.** Applying a higher-order substitution $\hat{\sigma} = \{\hat{v}_1 \mapsto \hat{t}_1, \ldots, \hat{v}_n \mapsto \hat{t}_n\}$ to a higher-order term $\hat{t}$ results in the simultaneous replacement of all occurrences in $\hat{t}$ of each variable (respectively functional variable) $\hat{v}_i$ by its corresponding higher-order term (respectively functional variable) $\hat{v}_i$ as defined by the higher-order substitution. This is denoted with the expression $\hat{\sigma}(\hat{t})$.

Definition 3.3.9 (Higher-Order Anti-unifier (and Unifier))**.** The substitution $\hat{\sigma}$ is a *higher-order unifier* for the higher-order terms $\hat{t}_1$ and $\hat{t}_2$ if and only if $\hat{\sigma}(\hat{t}_1) = \hat{\sigma}(\hat{t}_2)$. The higher-order term $\hat{t}$ is a *higher-order anti-unifier* (or generalization) for higher-order terms $\hat{t}_1$ and $\hat{t}_2$, if and only if there exist higher-order substitutions $\hat{\sigma}_1$ and $\hat{\sigma}_2$ such that $\hat{\sigma}_1(t) = \hat{t}_1$ and $\hat{\sigma}_2(t) = \hat{t}_2$.

As an example, a higher-order anti-unifier of the terms $f(a,b)$ and $g(a,b)$ is $F(a,b)$ as depicted in Figure 3.10. For simplicity, I henceforth drop the adjectival phrases "first-order" and "higher-order" in cases where the intent is clear from the context.

Applying higher-order anti-unification could help to construct 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 equivalent structures, missing structures, and different ordering of arguments.

3.4 Higher-order anti-unification modulo equational theories

In higher-order anti-unification modulo equational theories, a set of equational theories, which treat differing structures as equivalent, is defined to incorporate background knowledge. Each equational theory $=_E$ determines which terms are considered equal and a set of these equations can be applied on higher-order terms to determine structural equivalences; terms can then be replaced with alternative but equivalent terms. For example, I have introduced an equational theory $=_E$, such that $f(t, u) =_E f(u, t)$ to indicate that the ordering of arguments of f does not matter in our context.

I have also defined a set of equational theories to incorporate semantic knowledge of structural equivalences supported by the Java language specification, as it provides various syntactic ways to define semantically equivalent structures. For example, consider **for**- and **while**-statements that are two kinds of looping structure in Java: they have different syntax but cover the same concept. To be able to anti-unify these structures meaningfully, we need an equational theory that can equate any **for**-loop

$$\textbf{for}(\mathit{inits};\ \mathit{test};\ \mathit{updates})\ \{...\}$$

to an equivalent **while**-loop

$$\mathit{inits};\ \textbf{while}(\mathit{test})\ \{\textbf{try}\ \ \{...;\}\ \ \textbf{finally}\ \ \{\mathit{updates};\}\}.$$

3.4.1 Loss of uniqueness

Defining complex substitutions in higher-order anti-unification modulo theories results in losing the uniqueness of the MSA. For example, consider the terms $f_1(g(a, e))$ and $f_2(g(a, b), g(d, e))$. As described in Figure 3.11, two MSAs exist for these terms: we can anti-unify $g(a, e)$ and $g(a, b)$ to create the anti-unifier $g(a, X)$ and anti-unify $g(d, e)$ with the NIL structure to create the anti-unifier $G(Y, Z)$; or we can anti-unify $g(a, e)$ and $g(d, e)$ to create the anti-unifier $g(X, e)$ and anti-unify $g(a, b)$ with the NIL structure to create the anti-unifier $F(Y, Z)$.

$$\begin{tikzcd}
& \boxed{F(g(X,e),G(Y,Z))} \arrow[dl, "{\sigma_1 = \{F \mapsto f_2, X \mapsto d, G \mapsto g, Y \mapsto a, Z \mapsto b\}}"'] \arrow[dr, "{\sigma_2 = \{F \mapsto f_1, X \mapsto a, G \mapsto \mathsf{NIL}, Y \mapsto \mathsf{NIL}, Z \mapsto \mathsf{NIL}\}}"] & \\
\boxed{f_2(g(a,b),g(d,e))} & & \boxed{f_1(g(a,e))}
\end{tikzcd}$$

$$\begin{tikzcd}
& \boxed{F(g(a,X),G(Y,Z))} \arrow[dl, "{\sigma_1 = \{F \mapsto f_1, X \mapsto b, G \mapsto g, Y \mapsto d, Z \mapsto e\}}"'] \arrow[dr, "{\sigma_2 = \{F \mapsto f_2, X \mapsto e, G \mapsto \mathsf{NIL}, Y \mapsto \mathsf{NIL}, Z \mapsto \mathsf{NIL})}"] & \\
\boxed{f_2(g(a,b),g(d,e))} & & \boxed{f_1(g(a,e))}
\end{tikzcd}$$

Figure 3.11: Ambiguous higher-order anti-unification modulo theories of the terms $f_2(g(a,b),g(d,e))$ and $f_1(g(a,e))$, creating multiple MSAs.

Despite having multiple potential MSAs, I need to determine one single MSA that is the most appropriate in my 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 to variables in a term. Therefore, I need to use an approximation technique to construct one of the best MSAs that can sufficiently solve the problem.

3.5 Practical matters

One must be particularly careful about the equational theories that are introduced to avoid having the resulting framework be undecidable. Furthermore, searching through a large space of possible anti-unifiers to find the most specific one can have unacceptably poor runtime performance. Since our goal is to be able to extract commonalities out of Java source code, we approximate the notion of higher-order anti-unification modulo equational theories as follows.

3.5.1 Neutral elements and pumping

Sometimes it is desirable to anti-unify terms with differing numbers of arguments. To permit this, we can utilize the notion of pumping transformations and neutral elements. For example, with the binary function symbols “$+$” and “$\times$” we have the neutral elements “0” and “1” respectively, so that $+(x,0)=x$ and $\times(x,1)=x$ (also $+(0,x)=x$ and $\times(1,x)=x$ because of the commutativity of these functions). We can generalize this notion to the special n-ary function symbols $pump_n$ and to the special constant symbols ν_{pump_n} (each representing the neutral symbol for the indicated function), so that for any term t,

$$pump_n(\nu_{pump_n},\ldots,\nu_{pump_n},t,\nu_{pump_n},\ldots,\nu_{pump_n})=t,$$

where there are exactly n arguments to $pump_n(\cdot)$. Furthermore, we define $pump_n(\cdot)$ to be perfectly commutative with respect to its argument order, so that t may equivalently appear as any argument therein.

For simplicity of exposition, we use the label NIL in place of both the functional symbols $pump_n$ and the neutral element of each of the functions thereby defined, as the precise meaning can be interpreted from the context. Obviously, NIL representing an n-ary functional symbol cannot be meaningfully anti-unified with NIL representing an $n-k$-ary functional symbol—unless it is itself pumped.

This notion of pumping is equivalent to defining an equational theory $=_{pump}$ such that

$$\begin{gathered}
pump_n(\nu_{pump_n},\ldots,\nu_{pump_n},t,\nu_{pump_n},\ldots,\nu_{pump_n})\\
=_{pump}\ pump_{n-1}(\nu_{pump_{n-1}},\ldots,\nu_{pump_{n-1}},t,\nu_{pump_{n-1}},\ldots,\nu_{pump_{n-1}})\\
=_{pump}\ \cdots\\
=_{pump}\ pump_1(t)\\
=_{pump}\ t
\end{gathered}$$

$$\begin{tikzcd} & \boxed{F(X,b)} \arrow[dl, "\sigma_1 = (F \mapsto f, X \mapsto a)"'] \arrow[dr, "\sigma_2 = (F \mapsto \mathsf{NIL}, X \mapsto \mathsf{NIL})"] & \\ \boxed{f(a,b)} & & \boxed{\mathsf{NIL}(\mathsf{NIL}, b) =_{pump} b} \end{tikzcd}$$

Figure 3.12: Higher-order anti-unification modulo theories of the terms $f(a,b)$ and b.

Or equivalently:

$$\begin{aligned} &\mathsf{NIL}(\mathsf{NIL}, \ldots, \mathsf{NIL}, t, \mathsf{NIL}, \ldots, \mathsf{NIL}) \\ &=_{pump} \mathsf{NIL}(\mathsf{NIL}, \ldots, \mathsf{NIL}, t, \mathsf{NIL}, \ldots, \mathsf{NIL}) \\ &=_{pump} \cdots \\ &=_{pump} \mathsf{NIL}(t) \\ &=_{pump} t \end{aligned}$$

For simplicity, I refer to this as the NIL-theory.

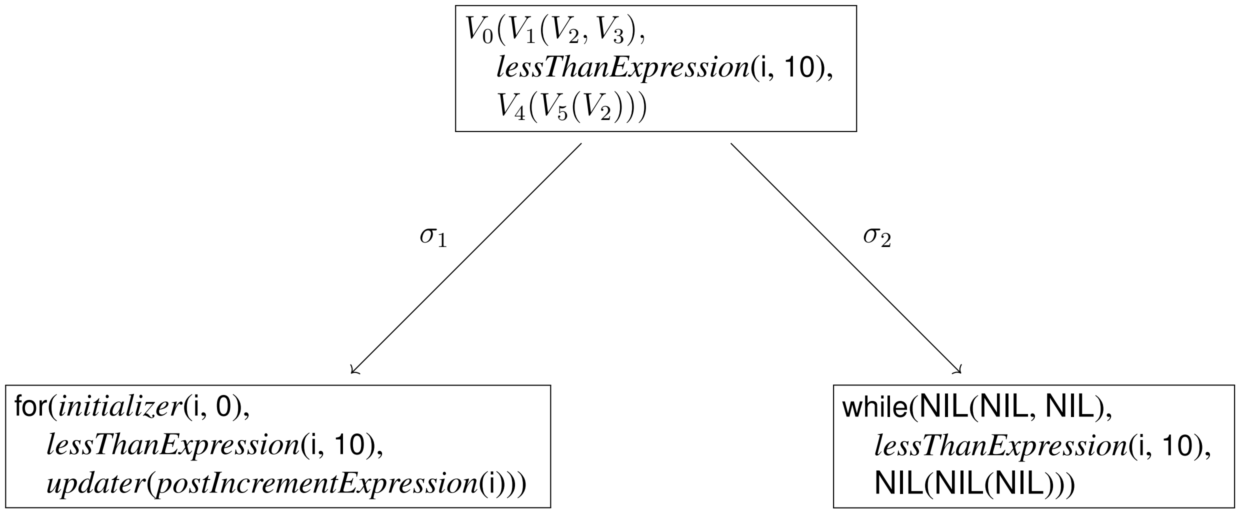
For example, we can anti-unify the two terms b and $f(a,b)$ through the application of the NIL-theory by creating the term $\mathsf{NIL}(\mathsf{NIL}, b)$—which is $=_{pump}$ to b—and anti-unifying $\mathsf{NIL}(\mathsf{NIL}, b)$ with $f(a,b)$ as depicted in Figure 3.12 to arrive at $F(X,b)$.

3.5.2 Variadic functions and overloading

I introduce an equational theory that treats two functions $f(\cdot)$ and $g(\cdot)$ as equivalent if their functional symbol is identical even if their arity differs. This captures the fact that identically named functions will generally represent minor variations on the same concept.

3.5.3 Loops

For simplicity, we treat **for**- and **while**-loops as functions with differing arguments, defining an equivalence equation $=_{loops}$ that allows their direct anti-unification. We then utilize the NIL-theory to handle the varying number of arguments as the **for**-loop has three arguments (actually, all three are optional in practice) whereas the **while**-loop only has one. Using the NIL-theory we can create the structure **while**(NIL(NIL, NIL), *lessThanExpression*(i, 10), NIL(NIL, NIL)) that is $=_{loops}$ to **while**(*lessThanExpression*(i, 10)) and construct the anti-unifier, $V_0(V_1(V_2, V_3)$, *lessThanExpression*(i, 10), $V_4(V_5(V_2)))$, as depicted in Figure 3.13.



$V_0(V_1(V_2, V_3),$ *lessThanExpression*(i, 10), $V_4(V_5(V_2)))$
σ_1
σ_2
for(*initializer*(i, 0), *lessThanExpression*(i, 10), *updater*(*postIncrementExpression*(i)))
while(NIL(NIL, NIL), *lessThanExpression*(i, 10), NIL(NIL(NIL)))

Figure 3.13: Anti-unification of the structures **for**(*initializer*(i, 0), *lessThanExpression*(i, 10), *updater*(*postIncrementExpression*(i))) and **while**(NIL(NIL, NIL), *lessThanExpression*(i, 10), NIL(NIL, NIL)). The substitutions are defined as follows: $\sigma_1 = (V_0 \mapsto \text{for}, V_1 \mapsto \textit{initializer}, V_2 \mapsto \text{i}, V_3 \mapsto 0, V_4 \mapsto \textit{updater}, V_5 \mapsto \textit{postIncrementExpression})$; and $\sigma_2 = (V_0 \mapsto \text{while}, V_1 \mapsto \text{NIL}, V_2 \mapsto \text{NIL}, V_3 \mapsto \text{NIL}, V_4 \mapsto \text{NIL}, V_5 \mapsto \text{NIL})$.

3.6 The Jigsaw tool

Jigsaw is a plug-in to the Eclipse integrated development environment (IDE), which was developed by Cottrell et al. [2008] to support small-scale source code reuse via structural correspondence. A small-scale reuse task can be divided into two phases. The first phase involves the developer identifying a source code snippet that implements functionality that is missing within a target system. The second phase involves integrating the source code snippet within the target system. Jigsaw supports the small scale reuse task by identifying structural correspondences between the code snippet and the context into which the code should be pasted, in order to suggest to developers what parts already exist within the target system, what parts are missing, and what parts need to be modified to fit into the context. In summary, the Jigsaw tool determines the structural correspondences between two Java source code fragments through the application of higher-order anti-unification modulo equational theories such that parts of one fragment can be integrated to the other one for small-scale code reuse.

In general, the approach of Cottrell et al. proceeds in three steps. First, it generates an augmented form of AST, called a *correspondence AST* (CAST), where each node holds a list of candidate correspondence connections, each implicitly representing an anti-unifier. To find candidate correspondences amongst the CASTs of the original system and the target system, it uses a similarity measure that relies on syntactic similarity along with simple knowledge of semantic equivalences supported by the Java language specifications. Although the CAST structure may represent many anti-unifiers, they used a greedy selection algorithm to select the best fit for each node via thresholding in order to approximate the optimal generalization. That is, the correspondence connections with a similarity value below a threshold are removed. Second, when there is more than one candidate correspondence connection for a node, the developer is prompted to resolve the conflict by selecting the best fit for his functionality. Third, the best correspondences are used to semi-automatically perform the integration task by replacing the developer's original source code with the integrated version. The Jigsaw tool is a proof-of-concept implementation of this approach.

Underlying the Jigsaw tool is the Jigsaw framework for determining likely structural correspondences between two ASTs; I simply refer to "Jigsaw" henceforth to intend the Jigsaw framework. The Jigsaw similarity function returns a value in $[0, 1]$ where zero indicates complete lack of similarity and one indicates perfect similarity. In general, this function returns a value above zero if the compared nodes are of identical type, and thus it returns a similarity of 0 for the nodes of different types. However, it uses several heuristics to improve the utility of the similarity measurement by defining an arbitrary value for the nodes that are syntactically different but are semantically relevant. For example, the similarity between names of AST nodes is measured using a normalized computation based on the length of the longest common substring. The comparison of the **int** and

long nodes is another example, where an arbitrary value of 0.5 is defined as the similarity, since they are not of syntactically identical types but have a semantic equivalence. This function also detects the correspondence between **for**-, enhanced-**for**-, **while**-, and **do**-loop statements; and **if** and **switch** conditional statements.

As I intend to construct a structural generalization from ASTs of two logged methods via structural correspondence, it could be helpful to use the first phase of the proposed approach to find candidate correspondences using the similarity measure. However, the second phase does not help determine the best correspondences needed in my context, as the CAST generated via thresholding neither resolves the conflicts that occur in constructing one single anti-unifier automatically, nor prevents the anti-unification of log statement nodes with any other nodes. There, the Jigsaw similarity function does not enable us to measure how similar are the usages of log statements inside methods. In addition, as the problem of this study is different, the integration phase of the approach is not related to my work. Instead, I should develop an algorithm to construct a detailed view of the generalization describing the structural commonalities and differences between logged methods. However, the CAST structure does not suffice to construct an anti-unifier: it does not allow the insertion of *structural variables* in place of nodes in the tree structure, and thus an extended form is required. In the following chapters, I will discuss my approach to create a structural generalization and its implementation by means of the higher-order anti-unification modulo theories.

3.7 Clustering

Clustering is the division of a collection of data objects into meaningful groups (clusters) [Jain et al., 1999]. The goal of clustering is to find groups of objects such that the objects in one cluster will be similar to one another and dissimilar from the objects in other clusters. The greater the similarity amongst the objects within a cluster and the greater the dissimilarity between the objects from various clusters are, the more distinct the clusters are [Tan et al., 2005]. In general, there are two major types of clustering:

- *Partitional clustering:* which aims to divide the data objects into non-overlapping clusters such that each data object is clearly in exactly one cluster.

- *Hierarchical clustering:* which aims to generate a set of nested clusters organized as a hierarchical structure. Each cluster in the structure is the union of its subclusters, and the cluster at the top contains all the data objects.

The following will introduce two popular techniques used to perform clustering on a data set:

3.7.1 K-means clustering algorithm

This algorithm is a partitional clustering approach that attempts to find a certain number of clusters, which are represented by centroids (the center of a cluster). In this algorithm, K is the number of the resulting clusters that has to be specified. Each cluster is associated with a centroid which is mostly the average of all the data objects within the cluster. The basic K-means clustering technique is described using Algorithm 3.1. This algorithm starts with K randomly selected data objects as initial centroids and then repeatedly assigns each data object to a cluster with the nearest centroid and computes the new clusters centroids accordingly. This process terminates when it reaches a state in which no objects are moving from one cluster to another, and thus, the centroids do not change. The K-means clustering algorithm is not a good fit to my purposes, as it requires one to specify the number of resulting clusters, which is not known here a priori.

Algorithm 3.1 Basic K-means clustering algorithm.

```
1: Start with K randomly selected data objects as initial centroids.
2: repeat
3:     Assign each object to its closest centroid.
4:     Update the centroid of each cluster.
5: until Centroids remain the same.
```

3.7.2 Agglomerative hierarchical clustering algorithm

The agglomerative hierarchical clustering algorithm produces a nested grouping of clusters, with single-point clusters at the bottom and an all-inclusive cluster at the top [Karypis et al., 1999]. Agglomerative hierarchical clustering is one of the mainstream clustering methods [Day and Edelsbrunner, 1984] that has applications in document retrieval [Voorhees, 1986] and information retrieval from a search engine query log [Beeferman and Berger, 2000]. Algorithm 3.2 describes the basic agglomerative hierarchical clustering approach. It starts with the individual objects as singleton clusters, and successively merges the two closest clusters until only one all inclusive cluster remains. In general, 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 ith and the jth clusters [Karypis et al., 1999].

Algorithm 3.2 Basic agglomerative hierarchical clustering algorithm.

```
1: Start with singleton clusters.
2: Compute the similarity matrix.
3: repeat
4:     Merge the closest cluster pair.
5:     Update the similarity matrix by computing the similarity between new cluster and all remaining clusters.
6: until Only one cluster remains.
```

There are various versions of agglomerative hierarchical clustering 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 two clusters. In the complete linkage method, the similarity is computed by the similarity of the farthest pair of data points of two clusters. In the average linkage method, the similarity is measured by the average of all pairwise similarities between data points of two clusters. In the centroids methods, each cluster is represented by a centroid, and the similarity between two clusters is measured by the similarity of the clusters' centroids. However, I need a modified version of the basic agglomerative hierarchical clustering algorithm to address my problem context: the merge and update operations should be terminated when the similarity between the closest clusters becomes below a pre-determined threshold value. Furthermore, I need a measure of similarity between clusters based on the similarity between their anti-unifiers. In

Chapter 6, I describe in detail the clustering algorithm that I applied to solve my problem.

3.8 Summary

I described abstract syntax trees (ASTs) as a standard syntactic representation of source code. Every AST can also be represented in a functional format (and vice versa) which constitute the standard theoretical concept of terms. I presented Eclipse JDT as a concrete framework that can be used to manipulate ASTs of a source code written in the Java programming language.

I demonstrated how the theoretical framework of anti-unification can be used as a technique to construct a common generalization of two given terms, and hence of two ASTs. First-order anti-unification permits terms to be replaced with variables and vice versa, but it is limited in that low-level commonality can be discarded due to high-level differences. Higher-order anti-unification overcomes this by permitting substitution relative to function symbols as well as terms. A further extension allows for arbitrary equational theories to embed knowledge of semantic equivalence. Unfortunately, this approach of higher-order anti-unification modulo theories leads to ambiguity and the potential for an infinite number of possible substitutions for every structural variable. To make use of that technique despite its weaknesses, we must apply an approximation technique to select amongst the best MSAs in order to reach a solution that is reasonable in practice. I also introduced Jigsaw, an existing framework for determining structural correspondences between ASTs and why it does not adequately address my problem. To address my problem, I describe in subsequent chapters how I extended the Jigsaw framework. Furthermore, I introduced the requirements on a hierarchical clustering algorithm that can be applied to classify logged methods into different categories.

Chapter 4

Determining Structural Correspondences

In order to construct a structural generalization describing the commonalities and differences between ASTs of two given logged methods (LMs), I first needed to find structural correspondences between their nodes. The approach to determining correspondences proceeds in three steps: (1) computing similarities between AST nodes (Section 4.1); (2) determining potential structural correspondences between nodes (Section 4.2); and (3) creating an extended form of AST to allow the insertion of variables in place of any nodes in the tree structure (Section 4.3).

To implement the approach, I needed a concrete framework for constructing and manipulating ASTs. The Eclipse integrated development environment provides such a framework in its Java Development Tools (JDT) component (as explained in Section 3.2). I also utilized Jigsaw [Cottrell et al., 2008], which is an existing framework for constructing an extended form of the AST structure, called CAST, where each node holds a list of candidate correspondence connections (as described in Section 3.6). Then, I extended the CAST structure to *anti-unifier ASTs* (AUASTs) structure that would allow the creation of an anti-unifier in a later step. To evaluate the effectiveness of Jigsaw to determining correspondences, I applied it on the ASTs of a sample set of logged Java methods and compared them in a pairwise manner. Section 4.4 describes my experimental setup, present my study, and discusses the results. I built atop this work in order to create anti-unifiers in a later phase.

4.1 Computing similarity between nodes

To compute similarity between AST nodes, I re-used a function developed by Cottrell et al. [2008], which first computes the number of common elements between the two nodes and then divides it by the size of the largest node, as described by the Equation 4.1.

$$similarity(A, B) = \frac{matches(A, B)}{\max\{|A|, |B|\}} \tag{4.1}$$

They compute the number of matched elements between the two ASTs A and B via the DETERMINE-MATCHES algorithm. As mentioned in Section 3.6, their approach only compares the nodes of identical or semantically equivalent types, thus the roots of both input ASTs are either leaf nodes or non-leaf nodes (line 2). If they are leaf nodes, the number of matches will be computed using the heuristics described in Section 3.6 (lines 3–4). If they are non-leaf nodes, the children of the two subtrees A and B are compared exhaustively in a pairwise manner (lines 5–16). For each child node of the subtree A, the highest matches with any child node of the subtree B is determined (lines 6–14). All these maximum matches are summed up and returned as the number of common elements, which can be used to determine a potential structural correspondence between the two nodes (lines 15–19) as will be described in the following section.

Algorithm 4.1 DETERMINE-MATCHES(A,B) computes the common elements ($matches$) between the two ASTs.

```
    DETERMINE-MATCHES(A,B)
 1: $matches \leftarrow 0$
 2: if LOOKUP-COMPARATOR($type[root[A]], type[root[B]]$) then
 3:     if $A$ instanceof Leaf-Node then
 4:         $matches \leftarrow$ MATCHES($A, B$)
 5:     else if $A$ instanceof Non-Leaf-Node then
 6:         for $childA \in children[A]$ do
 7:             $max \leftarrow 0$
 8:             for $childB \in children[B]$ do
 9:                 $matches \leftarrow$ DETERMINE-MATCHES($childA, childB$)
10:                 if $matches > max$ then
11:                     $max \leftarrow matches$
12:                 end if
13:             end for
14:         end for
15:         $matches \leftarrow matches + max$
16:     end if
17:     CREATE-CORRESPONDENCE-CONNECTION($A$,$B$,$matches$)
18: end if
19: return $matches$
```

4.2 Determining potential correspondences

As discussed in Section 3.6, the approach of Cottrell et al. [2008] generates an augmented form of AST, called a *correspondence AST* (CAST), where each node holds a list of candidate correspondence connections. The CREATE-CORRESPONDENCE-CONNECTION(A,B,$matches$) algorithm is used to create the CAST structure, which computes similarity between the two AST nodes using the Equation 4.1 (line 1). If the similarity value is above a pre-determined threshold, a correspondence connection will be created and added to the list of correspondence connections of the corresponding nodes (lines 2–6). As an example, Figure 4.1 shows potential structural correspondence connections created via the application of CREATE-CORRESPONDENCE-CONNECTION algorithm on the ASTs of Examples 1 and 2.

Algorithm 4.2 CREATE-CORRESPONDENCE-CONNECTION(A,B,$matches$) creates a candidate correspondence connection between the two AST nodes.

```
   CREATE-CORRESPONDENCE-CONNECTION($A$,$B$, $matches$)
1: $sim \leftarrow$ DETERMINE-MATCHES$(A, B)/\max\{|A|, |B|\}$
2: if $sim >$ Threshold then
3:     $corr \leftarrow$ NEW-CORRESPONDENCE-CONNECTION$(nodeA, nodeB, sim)$
4:     APPEND$(corrs[nodeA], corr)$
5:     APPEND$(corrs[nodeB], corr)$
6: end if
```

4.3 Constructing the AUAST

As described in Section 3.4, an anti-unified structure utilizes variables that must be substituted with proper substructures to regain the original structures. However, the CAST structure presented by Cottrell et al. [2008] would not allow the creation of an anti-unifier, as it does not contain any variables. Therefore, an extended form of it is required, the anti-unifier AST (AUAST), that would allow the insertion of variables in place of any node in the tree structure—including both subtrees and leaves—to indicate variations between the original structures.

To provide an example to demonstrate the structure, the anti-unified AUAST constructed from

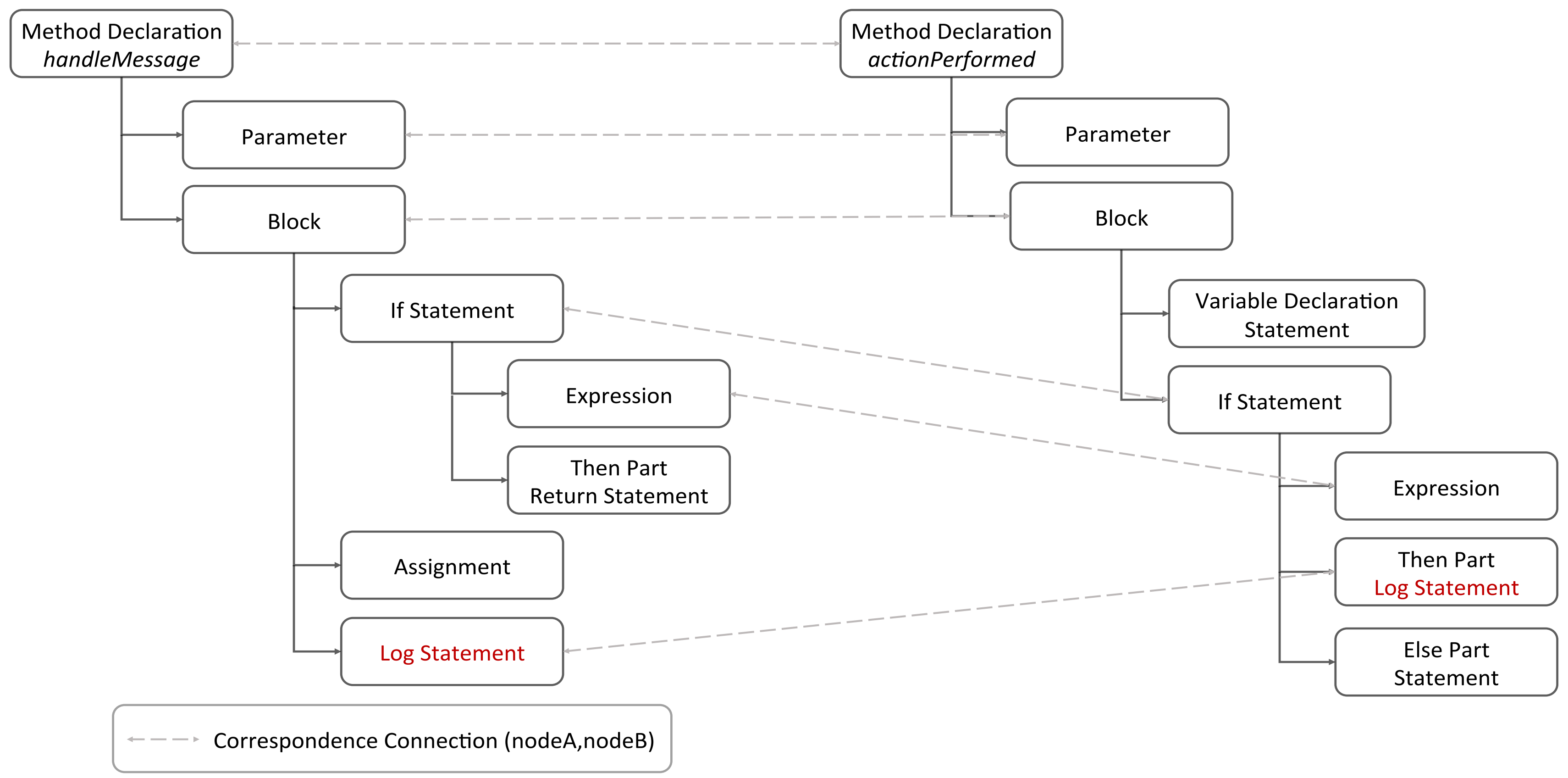


Figure 4.1: Simple CAST structure of the examples in Figures 3.5 and 3.6. The links between the CAST nodes indicate the structural correspondence connections.

the AUASTs of log statements in Figures 3.5 and 3.6 is depicted in Figure 4.2. The structural variables X and Y are used to indicate the structural variations, where the X structural variable refers to two simple names and the Y structural variable refers to two subtrees. The substitutions are defined in Equations 4.2 and 4.3.

$$\begin{aligned}\sigma_1 = (X \mapsto \textsf{WARNING}, Y \mapsto\ & \textit{additionExpression}(\\ & \textit{methodCall}(\textit{simpleName}(\textsf{getClassName}), \textit{arguments}()), \\ & \textit{stringLiteral}(\textsf{"should extend ..."}), \\ & \textit{methodCall}(\textit{simpleName}(\textsf{handleMessage}), \textit{arguments}()))) \end{aligned} \tag{4.2}$$

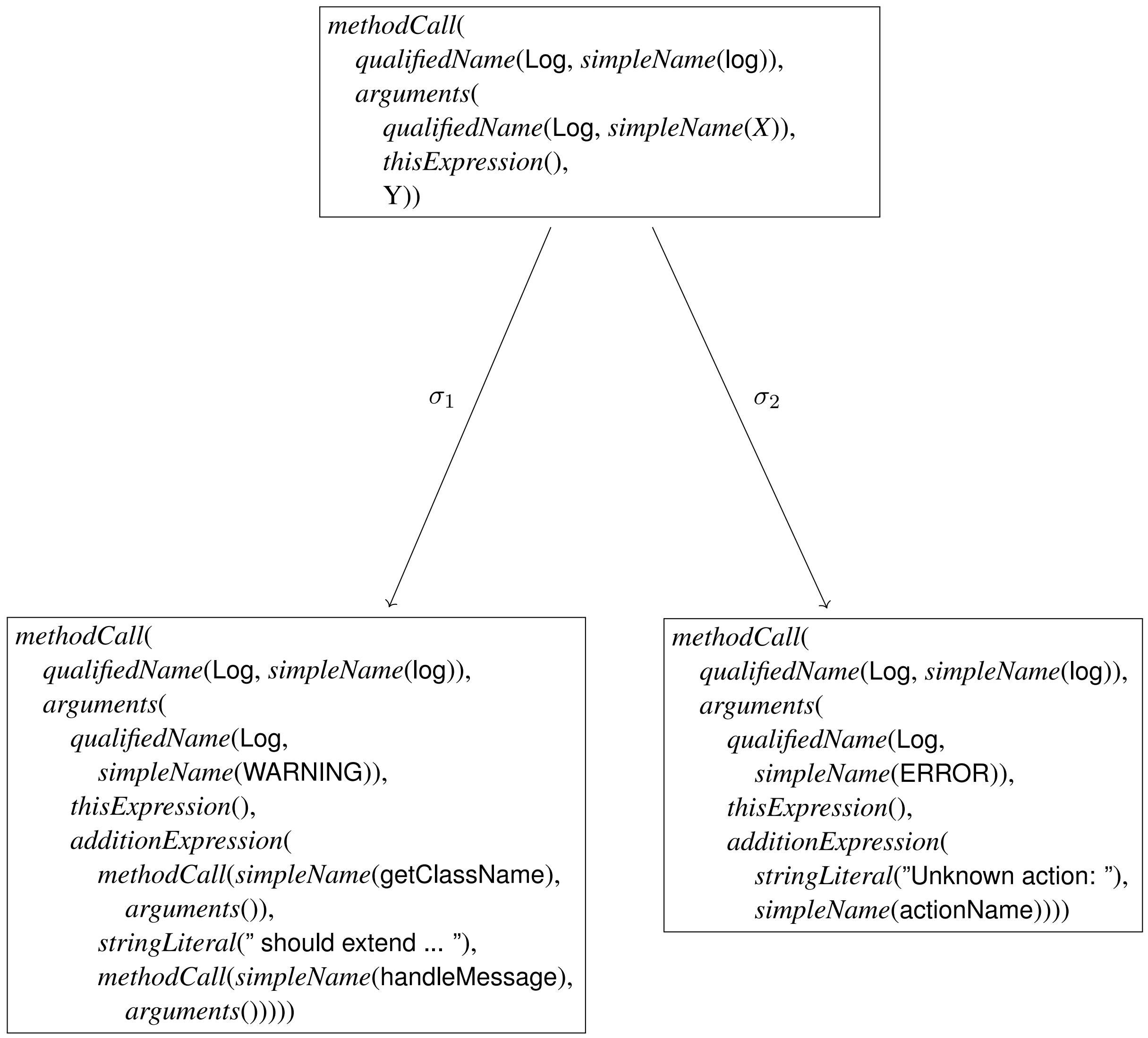
$$\begin{aligned}\sigma_2 = (X \mapsto \textsf{ERROR}, Y \mapsto\ & \textit{additionExpression}(\\ & \textit{stringLiteral}(\textsf{"Unknown action: "}), \\ & \textit{simpleName}(\textsf{actionName}))) \end{aligned} \tag{4.3}$$

4.4 An assessment of the Jigsaw framework

To validate the effectiveness of the approach of Cottrell et al. [2008], I applied the Jigsaw framework to compare the ASTs of a sample set of LMs in a pairwise manner: I constructed the CASTs of each pair. I selected the sample set from a real-world software system.

4.4.1 Setup

As the subject for my study, I used jEdit (version 4.2.15), a programmer's text editor tool written in the Java programming language. I chose this subject because it is a real program that is heavily used by many developers, and it employs real usage of log statements. My tool extracts all LMs within the source code of this program using the Eclipse JDT framework. I selected a subset



```
methodCall(
   qualifiedName(Log, simpleName(log)),
   arguments(
      qualifiedName(Log, simpleName(X)),
      thisExpression(),
      Y))
```

```
methodCall(
   qualifiedName(Log, simpleName(log)),
   arguments(
      qualifiedName(Log,
         simpleName(WARNING)),
      thisExpression(),
      additionExpression(
         methodCall(simpleName(getClassName),
            arguments()),
         stringLiteral(" should extend ... "),
         methodCall(simpleName(handleMessage),
            arguments())))))
```

```
methodCall(
   qualifiedName(Log, simpleName(log)),
   arguments(
      qualifiedName(Log,
         simpleName(ERROR)),
      thisExpression(),
      additionExpression(
         stringLiteral("Unknown action: "),
         simpleName(actionName))))
```

Figure 4.2: Anti-unification of the AUASTs of the log statements in Examples 1 and 2.

Case	Logged methods	Size (LOC)
1	PluginJAR.generateCache()	104
2	MiscUtilities .isSupportedEncoding(..)	9
3	EditBus.send(..)	14
4	EditBus.send(..)*	14
5	EditAction.Wrapper.actionPerformed(..)	5
6	EBPlugin.handleMessage(..)	6
7	BufferHistory.RecentHandler.doctypeDecl(..)	3
8	JARClassLoader.loadClass(..)	32
9	VFS.DirectoryEntry.RootsEntry.rootEntry(..)	36
10	ServiceManager.loadServices(..)	20

Figure 4.3: Logged methods used as our test suite. All are contained in the org.gjt.sp.jedit package with the exception of Case 9 that is in the org.gjt.sp.jedit.io package.

consisting of 9 LMs, showing varying levels of similarity on manual examination, as a test suite throughout this study (see Table 4.3). The EditBus.send(..) method contains two log statements. To handle this case I split it into two cases: Case 3 contained the first log statement while the second one was removed; Case 4 contained only the second log statement.

I describe my approach for LMs containing multiple log statements in detail in Section 5.4.2. The last three LMs were manually modified by adding some statements for the sake of dealing with important cases that we otherwise would have missed testing. Case 8 simulates the addition of an **if**-statement that formed a nested **if**-statement enclosing a log statement. Cases 9 and 10 simulate the addition of statements to improve the test coverage.

The Jigsaw framework was used to compare LMs of the test suite in a pairwise manner (55 test cases in total, including self-comparisons), producing the CASTs of each pair and measuring their Jigsaw similarity. I examined the generated CASTs of these test cases and selected, for the sake of this presentation, a subset of 4 test cases with various levels of correspondence as depicted in the Table 4.4. Case TC1 contains the comparison of a Java element with itself. Case TC2 contains the comparison of two Java elements that are both syntactically and semantically dissimilar. Case TC3 contains the correspondence between two Java elements that are syntactically dissimilar but are semantically relevant. Case TC4 contains the comparison of a log statement with another Java

Test case	Java source code fragment	Jigsaw similarity
TC1	Log.log(Log.WARNING,this,"Unknown action: " + actionName); Log.log(Log.WARNING,this,"Unknown action: " + actionName);	1

TC2	return entry;
	int i=0;
		0

TC3	**for** (**int** i=0; i < comps.length; i++) {...}
	while (entries.hasMoreElements()) {...}
		0.12

TC4	Log.log(Log.WARNING,this,"Unknown action: " + actionName);
	EditBus.removeFromBus(this);
		0.33

Figure 4.4: Results from examining the Jigsaw similarity for 4 sample Java source code fragment pairs.

element that is not a log statement but is syntactically relevant (they are both *methodInvocation* AST nodes).

4.4.2 Results

The results of the pairwise comparison between LMs of the test suite is visualized in Figure 4.5. As shown, the Jigsaw similarity for all self-comparisons is 1 as expected, while the level of Jigsaw similarity is different for pairs containing distinct LMs as per my manual examination.

The analysis of the 4 cases is shown in Table 4.4. Case 1 shows that a Java element that is compared with itself has a Jigsaw similarity of 1. Case 2 indicates that no correspondence connection is created when a Java element is compared with another Java element that is syntactically and semantically unrelated. Case 3 indicates that the similarity between a **for**-statement and a **while** -statement is non-zero, and that Jigsaw is able to detect semantic correspondences between Java elements for special cases where syntactic correspondence is absent. Case 4 shows that a logging call has non-zero similarity with another Java element that is not a logging call but is syntactically relevant. This case will be handled via the removal of this kind of correspondence connection, as will be described in Section 5.1.

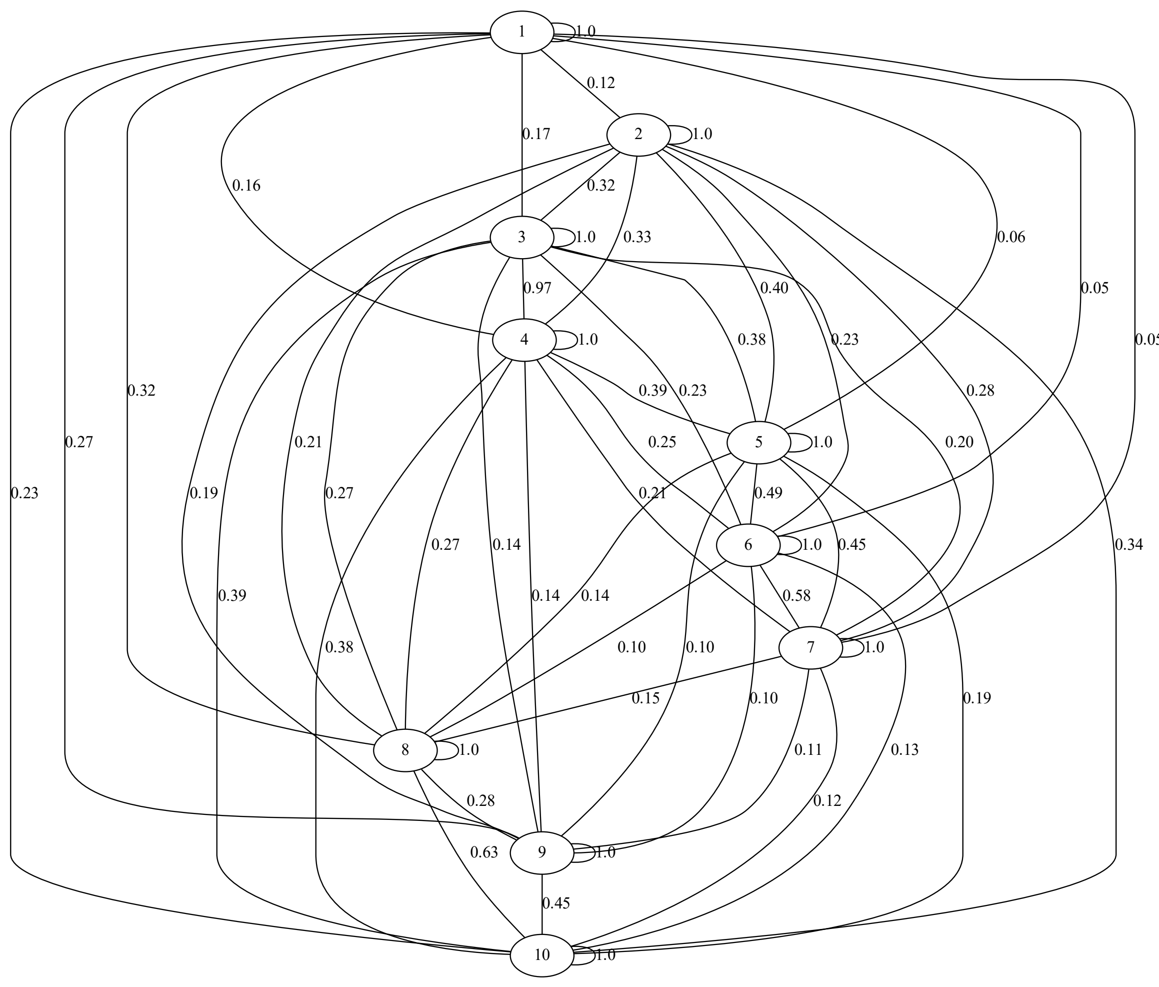


Figure 4.5: A similarity graph representing pairwise Jigsaw similarities between LMs shown in Table 4.3.

4.5 Summary

I described the approach proposed by Cottrell et al. [2008] to determining structural correspondences between AST nodes using a similarity measure. I assessed the functionality of their approach to address the problem of this study through an empirical study that uses the Eclipse JDT framework for extracting the ASTs of a sample set of LMs selected from a real-world software system, and applies the Jigsaw framework for constructing the correspondence structures.

Chapter 5

Constructing Structural Generalizations

I now consider how Jigsaw and its CAST structures could help me (1) to construct an approximation of the best anti-unifier to my problem from the AUASTs of two logged methods with special attention to log statements and (2) to develop a similarity measure between the two structures, which can provide us with useful information for clustering a set of logged methods in a later phase. The constructed anti-unifier can be viewed as a generalization that represents the structural commonalities and differences between the two logged methods.

To approximate the best anti-unifier for my problem, I develop a greedy selection algorithm that determines the best correspondence for each node. My approach contains a sequence of 2 actions to find the best correspondences between two AUASTs: (1) it applies two constraints to prevent the anti-unification of log statement nodes with any other nodes; and (2) it determines the best correspondence for each AUAST node with the highest similarity and then removes the other correspondence connections involving those nodes (Section 5.1). However, to construct an anti-unifier, a further step is taken: the anti-unification of each AUAST node with its best correspondence (Section 5.3). Furthermore, I developed an algorithm to measure similarity between the usage of log statements in methods based on the selected correspondences (Section 5.2). An overview of the proposed approach is shown in Figure 5.1.

To evaluate my approach, I developed the anti-unifier-building tool atop Jigsaw, and conducted an experimental study on the test suite introduced in Section 4.4.1. In Section 5.4, I discuss the algorithms and decisions I made for implementing the anti-unifier-building tool and constructing a detailed view of the structural generalization. I also describe my experimental setup, present the study, and discuss the results in Section 5.5.

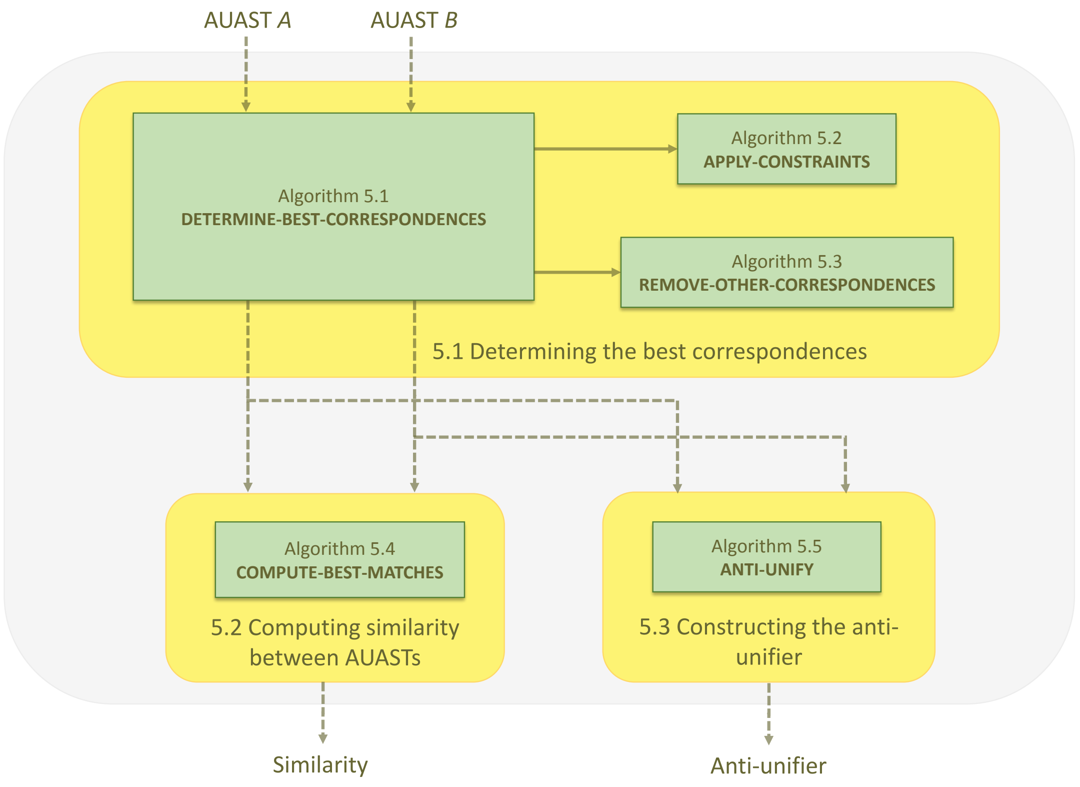

AUAST *A*
AUAST *B*
Algorithm 5.1
DETERMINE-BEST-CORRESPONDENCES
Algorithm 5.2
APPLY-CONSTRAINTS
Algorithm 5.3
REMOVE-OTHER-CORRESPONDENCES
5.1 Determining the best correspondences
Algorithm 5.4
COMPUTE-BEST-MATCHES
5.2 Computing similarity between AUASTs
Algorithm 5.5
ANTI-UNIFY
5.3 Constructing the anti-unifier
Similarity
Anti-unifier

Figure 5.1: Overview of the generalization process.

5.1 Determining the best correspondences

As discussed in Section 3.4, in general, higher-order anti-unification modulo theories has been proven to be undecidable. Therefore, to find one single anti-unifier that is an approximation of the optimal fit to my problem, I developed the DETERMINE-BEST-CORRESPONDENCES algorithm that greedily selects the most similar correspondence as the best fit for each AUAST node. Hence, each AUAST node can either be anti-unified with its best correspondence in the other AUAST or with "nothing". This algorithm 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 descending order based on the similarity measure (lines 1–9). However, to prevent the anti-unification of log statement nodes with any other nodes, I apply some constraints on the selection of correspondence connections prior to determining the best ones via the APPLY-CONSTRAINTS algorithm (line 8). Then the correspondence connection with the highest similarity value is determined as the best fit for the two nodes involved, and all other correspondence connections involving these nodes are removed using REMOVE-OTHER-CORRESPONDENCES algorithm (lines 10–14). This process terminates when no more correspondence connections are left in the list.

Algorithm 5.1 DETERMINE-BEST-CORRESPONDENCES(A) takes in the one of the AUASTs and determines the best correspondence connection with the highest similarity for each node.

```
    DETERMINE-BEST-CORRESPONDENCES($A$)
 1: $list \leftarrow ()$
 2: $nodes \leftarrow$ VISITOR($A$)
 3: for $node \in A$ do
 4:     for $corr \in corrs[node]$ do
 5:         APPEND($corr, list$)
 6:     end for
 7: end for
 8: APPLY-CONSTRAINTS($list$)
 9: SORT($list$)
10: for $corr \in list$ do
11:     $bestCorr[nodeA] \leftarrow corr$
12:     $bestCorr[nodeB] \leftarrow corr$
13:     REMOVE-OTHER-CORRESPONDENCES($corr, list$)
14: end for
```

To construct an anti-unifier of the AUASTs of logged methods with a focus on log statements, some constraints should be applied in determining correspondences. The first constraint (as described below) should be applied to prevent the anti-unification of log statement nodes with any other types of nodes.

Constraint 1. A log statement should either be anti-unified with another log statement or should be anti-unified with "nothing".

This constraint creates a further constraint:

Constraint 2. A structure containing a log statement should be anti-unified with a corresponding structure containing another log statement or should be anti-unified with "nothing".

As an illustration, consider the CASTs of the two examples in Figure 4.1. There is a candidate correspondence connection between the two log method invocation nodes and another between the two **if** statements. 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 anti-unified together. On the other hand, a correspondence connection is created between the two **if** statements; however, anti-unification of these statements includes anti-unifying their children nodes as well. Thus, statements inside the body of the **if** statements must be anti-unified with each other, indicating that log method invocation inside the body of **if** statement in the second example should be anti-unified 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. My approach applies these constraints by taking the following steps prior to determining the best correspondences:

1. Augment the AUAST node with a property to mark log statement nodes and structures enclosing them as "logged".

2. Remove correspondence connections where one node is marked as "logged" and the corresponding node is not via the APPLY-CONSTRAINTS algorithm.

The APPLY-CONSTRAINTS algorithm takes the list of correspondence connections, and removes the ones involving two nodes where one is "logged" and the corresponding node is not (lines 1–9).

Algorithm 5.2 APPLY-CONSTRAINTS($list$) applies the constraints on the list of correspondence connections.

```
    APPLY-CONSTRAINTS($list$)
 1: for $corr \in list$ do
 2:     $nodeA \leftarrow nodeA[corr]$
 3:     $nodeB \leftarrow nodeB[corr]$
 4:     if (($nodeA$ instanceof Logged) & ($nodeB$ instanceof Non-Logged)) or
 5: (($nodeA$ instanceof Non-Logged) & ($nodeB$ instanceof Logged)) then
 6:         REMOVE($corr$, $list$)
 7:         REMOVE($corr$, $corrs[nodeA]$)
 8:         REMOVE($corr$, $corrs[nodeB]$)
 9:     end if
10: end for
```

The 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 correspondence connections of the first node involved in the connection (line 6 and line 13); and the list of correspondence connections of the second node involved in the connection (line 7 and line 14). As an example, Figure 5.2 shows the correspondence connections between AUAST nodes after applying the DETERMINE-BEST-CORRESPONDENCE algorithm on the list of potential correspondence connections.

Algorithm 5.3 REMOVE-OTHER-CORRESPONDENCES($corr$,$list$) removes all other correspondences involving the nodes of a particular correspondence connection ($corr$) from the lists of correspondences.

```
    REMOVE-OTHER-CORRESPONDENCES($corr$, $list$)
 1: $listA \leftarrow corrs[nodeA[corr]]$
 2: $listB \leftarrow corrs[nodeB[corr]]$
 3: for $corrA \in listA$ do
 4:     if $corrA \neq corr$ then
 5:         REMOVE($corrA$, $listA$)
 6:         REMOVE($corrA$, $corrs[nodeA[corrA]]$)
 7:         REMOVE($corrA$, $corrs[nodeB[corrA]]$)
 8:     end if
 9: end for
10: for $corrB \in listB$ do
11:     if $corrB \neq corr$ then
12:         REMOVE($corrB$, $listB$)
13:         REMOVE($corrB$, $corrs[nodeA[corrB]]$)
14:         REMOVE($corrB$, $corrs[nodeB[corrB]]$)
15:     end if
16: end for
```

5.2 Computing similarity between AUASTs

Similarity computation is particularly important for the clustering phase that relies on accurate estimation of similarity between logged methods (see Chapter 6). 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 another. The utility of a similarity function

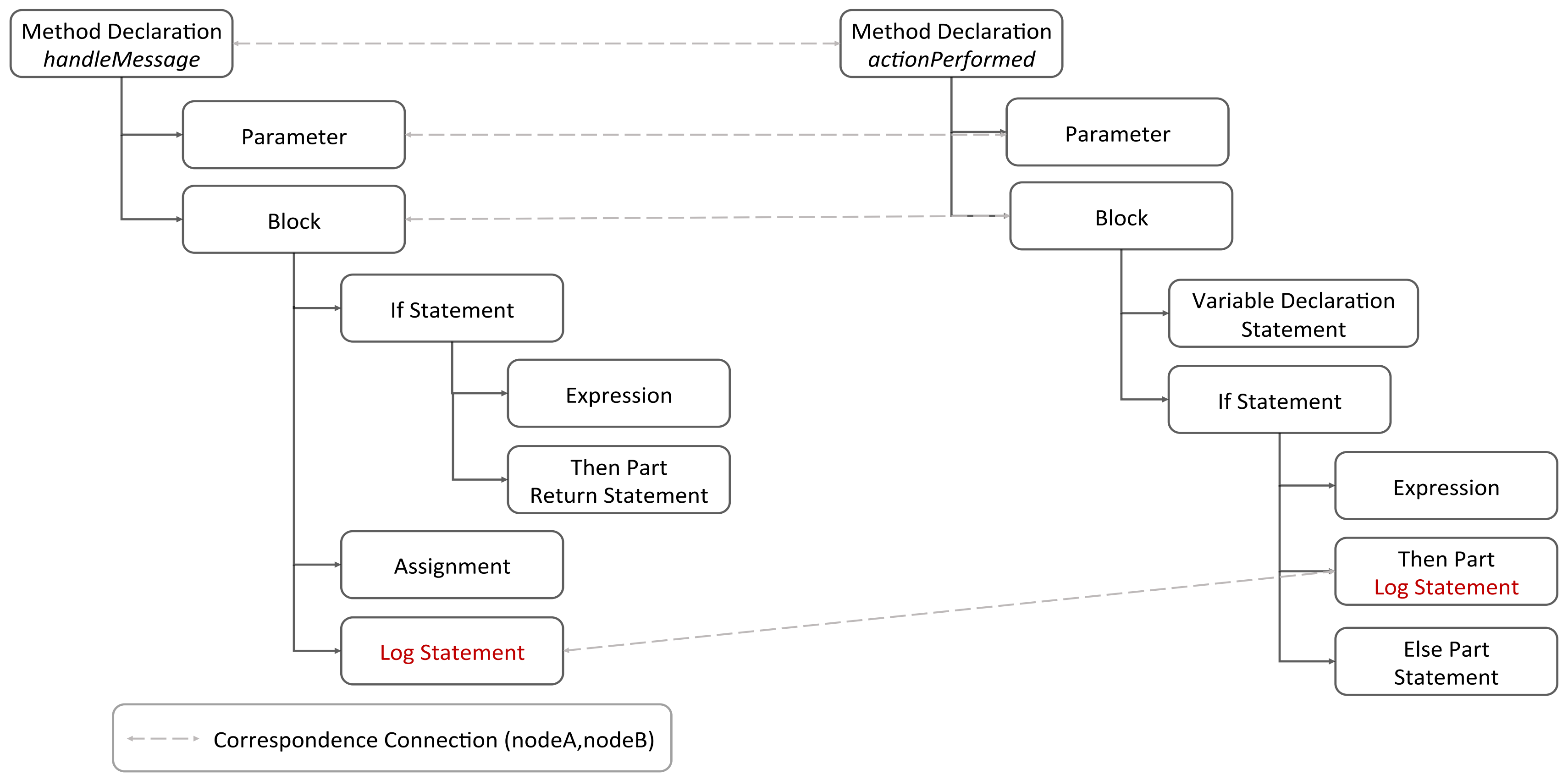


Figure 5.2: Simple AUAST structure of examples in Figures 3.5 and 3.6. The links between AUAST nodes indicate structural correspondences selected as the best fit using the DETERMINE-BEST-CORRESPONDENCES algorithm.

can be determined based on how well it enables us to produce accurate results for a particular task. In this study, a similarity measure is needed to classify logged methods based on the structural similarity in the usage of log statements. The similarity for my application can be defined as the number of common elements over the total number of elements of the anti-unifier constructed based on the selected correspondences from the previous step (described by the Equation 5.1).

$$similarity(A, B) = \frac{matches(A, B)}{|A| + |B|} \tag{5.1}$$

The similarity between two AUAST nodes is computed by dividing the number of matched elements among the two nodes by the size of the largest node (Equation 4.1). The number of matches between the two AUASTs A and B is computed via the COMPUTE-BEST-MATCHES algorithm. If the two AUASTs are leaf nodes, the number of matches is computed (lines 2–3) using the heuristics proposed by Cottrell et al. [2008], previously described in Section 3.6. Otherwise, the best correspondences between the two subtrees are selected using the DETERMINE-BEST-CORRESPONDENCES algorithm, and the matches between each children of the subtree A and its best corresponding node in the subtree B is computed and propagated to the parent node (lines 4–12).

Algorithm 5.4 COMPUTE-BEST-MATCHES(A,B) computes the matches between the two ASTs based on the best correspondences.

```
    COMPUTE-BEST-MATCHES(A,B)
 1: $matches \leftarrow 0$
 2: if $A$ instanceof Leaf-Node then
 3:     $matches \leftarrow$ MATCHES$(A, B)$
 4: else if $A$ instanceof Non-Leaf-Node then
 5:     DETERMINE-BEST-CORRESPONDENCES($A$, $B$)
 6:     for $childA \in children[A]$ do
 7:         if $bestCorr[childA] \neq$ NULL then
 8:             $matches \leftarrow matches +$ COMPUTE-BEST-MATCHES$(childA, nodeB[bestCorr[childA]])$
 9:         end if
10:     end for
11: end if
12: return $matches$
```

5.3 Constructing the anti-unifier

Once the best correspondences have been determined between two AUASTs, I construct a new anti-unified AUAST by traversing the original AUAST structures recursively and anti-unifying each node with its best correspondence. The new anti-unified structure is a generalization of two original structures where common structural nodes are represented by copies and differences are represented by structural variables. The variables may be inserted in place of any node in the AUAST, including both subtrees and leaves, and can be substituted with proper original substructures to regain the original structures.

Anti-unification of two AUASTs is performed via the ANTIUNIFY algorithm. If the two AUASTs are leaf nodes, the anti-unifier will be created through the anti-unification of the two nodes (lines 2–3). Otherwise, the anti-unified subtree is created by anti-unifying each child of one subtree with its best corresponding node in the other subtree. If there is no correspondence for a node, the anti-unified node will be created through the anti-unification of the node with the NIL structure. All these anti-unified nodes are appended to the list of children of the anti-unifier (lines 4–21). The details of constructing a detailed view of the anti-unifier for my application will be discussed in Section 5.4.1.

5.4 The anti-unifier building tool

The anti-unifier building tool is a proof-of-concept implementation of my anti-unification approach, which is developed atop the Jigsaw framework to construct a detailed view of structural generalization. To create the AUAST structure using Eclipse JDT framework that addresses the limitations of CAST to constructing an anti-unifier, I added the following structural properties:

- *Simple structural variable properties*: an extension of simple structural properties referring to two simple values to allow the insertion of variables in place of leaves.

- *Child structural variable properties*: an extension of child structural properties referring to

Algorithm 5.5 ANTIUNIFY(A,B) creates the anti-unifier of two AUASTs through the anti-unification of each node with its best correspondence.

```
ANTIUNIFY($A$, $B$)
1:  $antiunifier \leftarrow$ NULL
2:  if $A$ instanceof Leaf-Node then
3:      $antiunifier \leftarrow$ CONSTRUCT-ANTIUNIFIER$(A, B)$
4:  else if $A$ instanceof Non-Leaf-Node then
5:      for $childA \in children[A]$ do
6:          if $bestCorr[childA] =$ NULL then
7:              $child \leftarrow$ ANTIUNIFY$(childA,$ NIL$)$
8:          else
9:              $child \leftarrow$ ANTIUNIFY$(childA, nodeB[bestCorr[childA]])$
10:         end if
11:         APPEND$(child, children[antiunifier])$
12:     end for
13: end if
14: if $B$ instanceof Non-Leaf-Node then
15:     for $childB \in children[B]$ do
16:         if $bestCorr[childB] =$ NULL then
17:             $child \leftarrow$ ANTIUNIFY$(childB,$ NIL$)$
18:         end if
19:         APPEND$(child, children[antiunifier])$
20:     end for
21: end if
22: return $antiunifier$
```

two child AST nodes to allow the insertion of variables in place of subtrees.

I also needed to make additional algorithms and decisions to construct a detailed generalization view suited to my application, which will be described in the following sections.

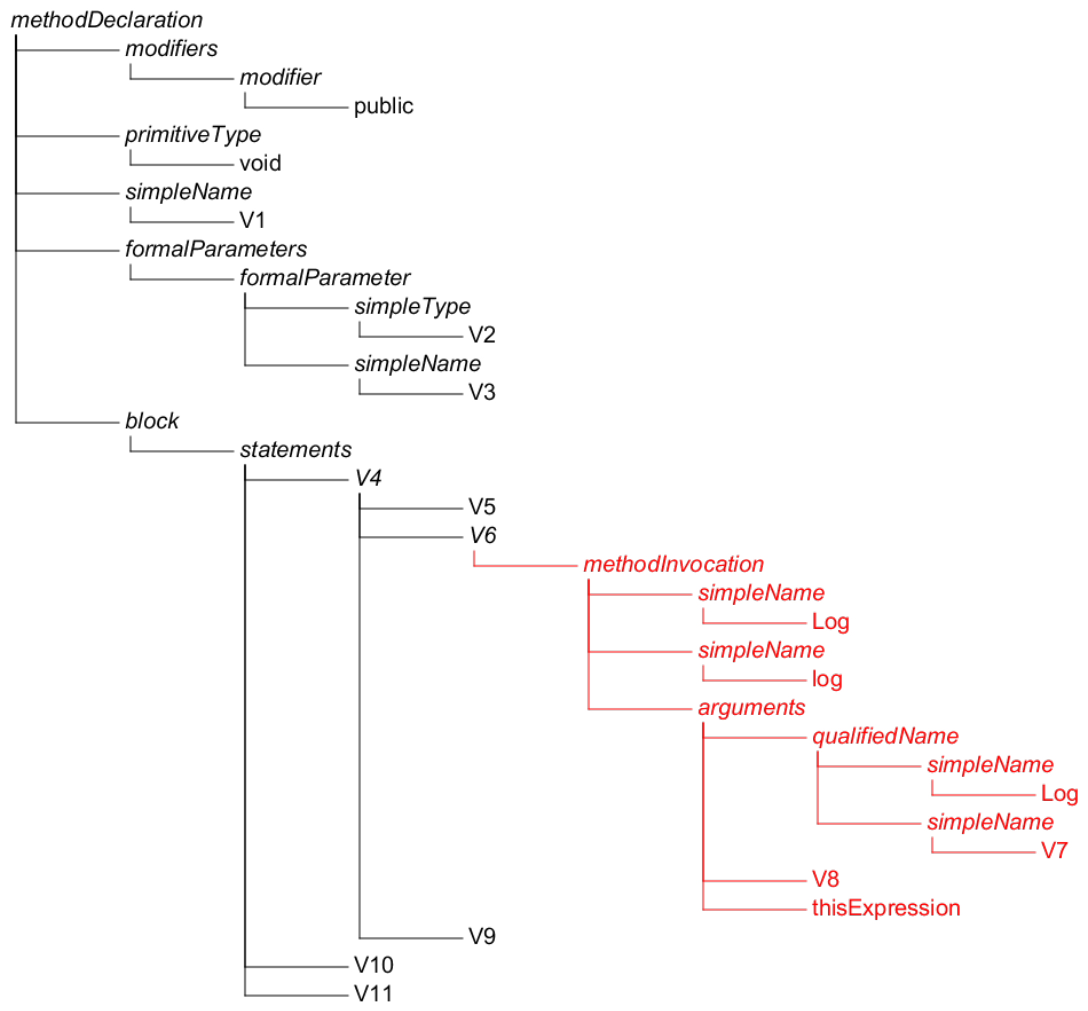
5.4.1 Constructing the detailed generalization view

To figure out the structural commonalities and differences amongst LMs, I developed an algorithm to construct a generalization view of the anti-unifier. Figure 5.3 shows the detailed view of the anti-unifier constructed from the AUASTs of LMs of Examples 1 and 2, generated by ELUS.

To create the detailed view of the structural generalization from two given AUASTs, I applied the ANTIUNIFY algorithm (Section 5.3) on the two AUAST nodes a and b through the anti-unification of their structural properties, as the Eclipse JDT utilizes these properties to record structural information of each Java element (Section 3.2). That is, for each structural property of the two nodes, if the property is common between them, a copy of it will be created and added to the structural properties of the anti-unifier; otherwise, a variable structural property is constructed referring to two property values and added to the anti-unifier's structural properties.

5.4.2 Methods containing multiple log statements

There might be some cases in which our approach is not able to anti-unify log statements in two input seeds, when there is more than one logging call in an LM. For example, consider the LMs in Figures 5.4 and 5.5. Figure 5.6 shows the simple AUASTs for these examples and potential correspondence connections between the AUAST nodes. Figure 5.7 shows the correspondence connections selected as the best match using our greedy algorithm. To anti-unify If Statement 1 with If Statement 3, we should anti-unify their structural properties. Thus, Log Statement 1 should be anti-unified with Log Statement 3, and Log Statement 4 should be anti-unified with NIL since there is no corresponding log statement in the body of If Statement 1, while there is a corresponding node for Log Statement 4 in the body of If Statement 2 (Log Statement 2).



```
Theta1 = {V1 -> actionPerformed, V2 -> ActionEvent, V3 -> evt, V4 -> ifStatement, V5 -> infixExpression(simpleName(action),oper
ator(==),nullLiteral()), V6 -> block, V7 -> ERROR, V8 -> infixExpression(stringLiteral("Unknown action: "),operator(+),simpleNa
me(actionName)), V9 -> methodInvocation(simpleName(context),simpleName(invokeAction),arguments(simpleName(evt),simpleName(actio
n))), V10 -> variableDeclarationStatement(simpleType(EditAction),fragments(variableDeclarationFragment(simpleName(action),metho
dInvocation(simpleName(context),simpleName(getAction),arguments(simpleName(actionName)))))), V11 -> NIL}

Theta2 = {V1 -> handleMessage, V2 -> EBMessage, V3 -> message, V4 -> NIL, V5 -> NIL, V6 -> NIL, V7 -> WARNING, V8 -> infixExpre
ssion(methodInvocation(simpleName(getClassName)),operator(+),stringLiteral(" should extend"),extendedOperands(stringLiteral(" E
ditPlugin not EBPlugin since it has an empty"),stringLiteral(" handleMessage()"))), V9 -> NIL, V10 -> ifStatement(simpleName(se
enWarning),returnStatement()), V11 -> assignment(simpleName(seenWarning),operator(=),booleanLiteral(true))}
```

Figure 5.3: The detailed view of the anti-unifier generated by ELUS from the AUASTs of Examples 1 and 2.

```
public void method1(){
        ...
        if (condition1){
                Log.log();
        }
        ...
        if (condition2){
                Log.log();
        }
        ...
}
```

Figure 5.4: A Java method that utilizes multiple log statements.

```
public void method2(){
        ...
        if (condition3){
                Log.log();
                Log.log();
        }
        ...
}
```

Figure 5.5: Another Java method that utilizes multiple log statements.

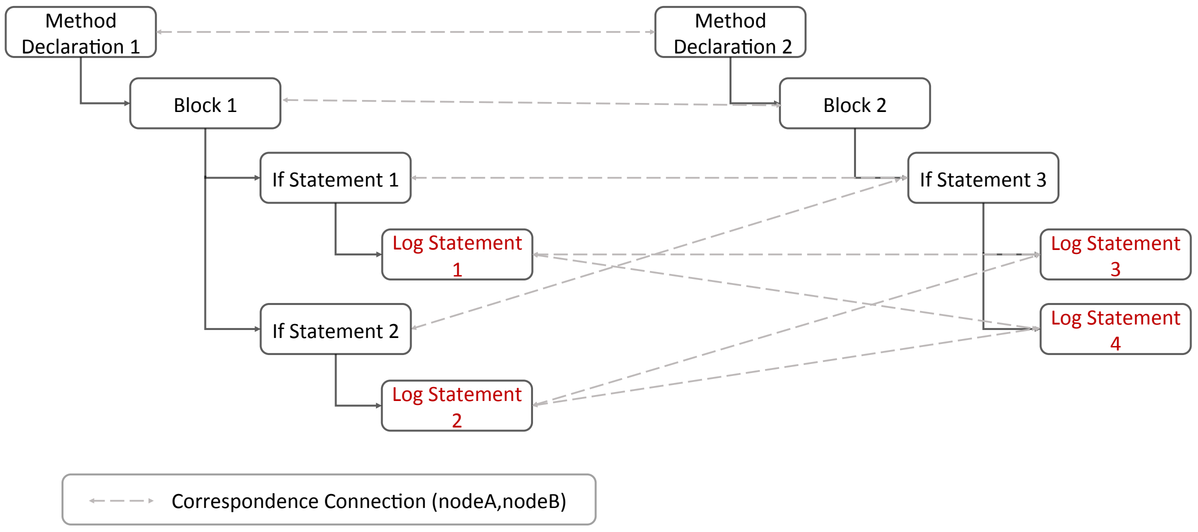


Figure 5.6: Simple AUAST structure of examples in Figures 5.4 and 5.5. Links between AUAST nodes indicate candidate structural correspondences detected by the Jigsaw framework.

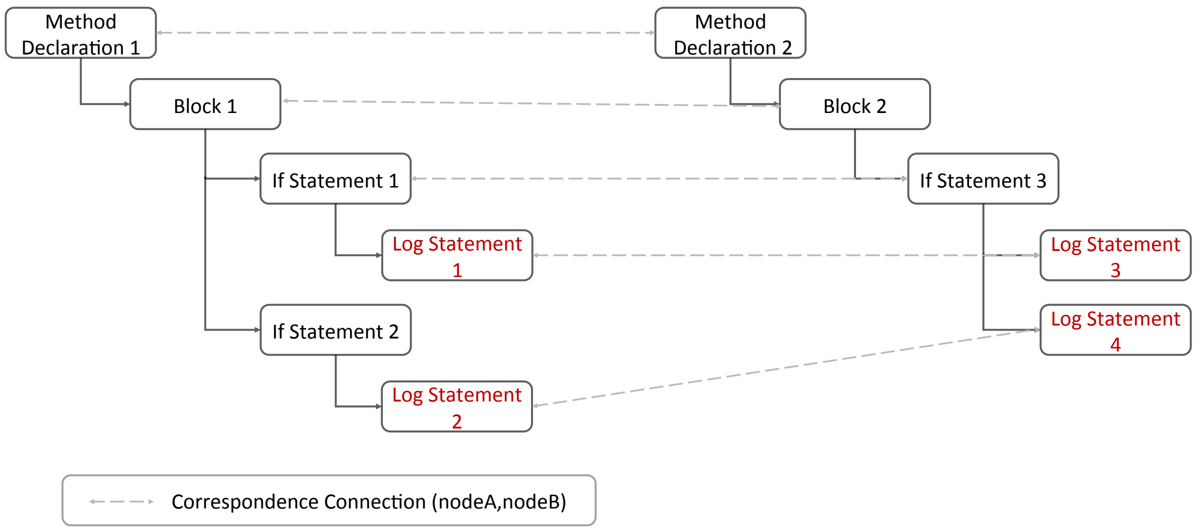


Figure 5.7: Simple AUAST structure of examples in Figures 5.4 and 5.5. Links between AUAST nodes indicate structural correspondences selected as the best match using the greedy algorithm.

To handle these cases, we can split them into more than one case, where each LM contains only one log statement. To do so, we need to create a copy of the LM for each log statement by maintaining that logging call and removing the other ones. For example, we need to create two copies for each logged Java method of examples in Figures 5.4 and 5.5 as depicted in Figures 5.8 and 5.9, respectively.

```
public void method1(){
        ...
        if(condition1){
                Log.log();
        }
        ...
        if(condition2){
                //removed
        }
        ...
}
```

```
public void method1(){
        ...
        if(condition1){
                //removed
        }
        ...
        if(condition2){
                Log.log();
        }
        ...
}
```

Figure 5.8: Creating multiple copies of the LM in Figure 5.4 for each log statement.

```
public void method2(){
        ...
        if(condition3){
                //removed
                Log.log();
        }
        ...
}
```

```
public void method2(){
        ...
        if(condition3){
                Log.log();
                //removed
        }
        ...
}
```

Figure 5.9: Creating multiple copies of the LM in Figure 5.5 for each log statement.

5.5 Evaluation

To assess the effectiveness of my anti-unification algorithm and the supporting tool, I conducted an experiment on the test suite described in Section 4.4.1.

5.5.1 Setup

In this study, I manually attempted to create the detailed anti-unifier view for each pair of LMs in the test suite (55 test cases in total). I first identified corresponding and non-corresponding Java elements for each LM pair with a focus on preventing the correspondence of log statements with anything else and then represented the anti-unifier in the detailed view (i.e., formatted as in Figure 5.3). I also computed the ratio of common Java elements in the detailed anti-unifier view to total number of Java elements of the two LMs to measure the similarity. I also ran the anti-unifier-building tool on each pair of LMs to construct the detailed anti-unifier view for each pair with special attention to log statements and to measure the similarity between the LM pairs. Furthermore, I used EclEmma, which is a Java code coverage tool for Eclipse, to measure the test coverage. Test coverage is defined as a measure of the completeness of the set of test cases.

5.5.2 Results

I present the results of my analysis for a subset of 10 test cases (see Table 5.10) in Table 5.11. The analysis of the output has been divided into two categories: correspondence and similarity. "Correspondence" refers to the number of corresponding lines-of-code (LOC) detected by my tool that were found to be corresponded by my manual examination as well, and the number of LOC were not detected as corresponded by my tool but were found to be corresponded in manual inspection. I also present the percentage of the correct corresponding LOC to the total number of LOC of the two LMs. "Similarity" refers to the similarity that is computed based on the detected correspondences. It is calculated using both my tool and manual experiment.

In Case 8, the `rootEntry (..)` method contains a nested **if**-statement enclosing a log statement

Test case	Logged methods
1	PluginJAR.generateCache() PluginJAR.generateCache()

2	PluginJAR.generateCache()
	EditBus.send(..)*

3	MiscUtilities.isSupportedEncoding(..)
	EditBus.send(..)

4	EditBus.send(..)
	EditBus.send(..)*

5	EditBus.send(..)*
	EditAction.Wrapper.actionPerformed(..)

6	EditBus.send(..)*
	BufferHistory.RecentHandler.doctypeDecl(..)

7	EditAction.Wrapper.actionPerformed(..)
	JARClassLoader.loadClass(..)

8	EditAction.Wrapper.actionPerformed(..)
	VFS.DirectoryEntry.RootsEntry.rootEntry(..)

9	PluginJAR.generateCache()
	BufferHistory.RecentHandler.doctypeDecl(..)

10	VFS.DirectoryEntry.RootsEntry.rootEntry(..)
	ServiceManager.loadServices(..)

Figure 5.10: 10 sample logged Java method pairs used as test cases; all are contained in the org.gjt.sp.jedit package with the exception of cases 8 and 10 that are in the org.gjt.sp.jedit.io package.

Test case	Correspondence		Similarity	
	Correct (%)	Incorrect	human	tool
1	104 (100)	0	1.0	1.0
2	8 (100)	0	0.13	0.13
3	6 (85)	1	0.19	0.16
4	4 (100)	0	0.29	0.29
5	5 (100)	0	0.21	0.21
6	3 (100)	0	0.2	0.2
7	5 (100)	0	0.11	0.11
8	7 (100)	0	0.1	0.1
9	3 (100)	0	0.03	0.03
10	14 (87)	2	0.27	0.22

Figure 5.11: Results of constructing anti-unifiers with a focus on log statements for the test cases presented in Table 5.10

and the actionPerformed(..) method contains an **if**-statement enclosing another log statement. The analysis showed that a correct correspondence was detected between the inner **if**-statement inside the nested **if**- and the single **if**-statement. Cases 3 and 10 contain statements that are not found to correspond by my tool even though correspondences exist. For example, in Case 3, the isSupportedEncoding(..) method contains an assignment statement enclosed by an **if**-statement that does not have any correspondences and the send(..) method contains another assignment statement inside a **for**-statement without any correspondences. However, no correspondence was detected between the two assignment statements since their parent nodes do not correspond.

The results of the pairwise comparison between LMs of the test suite is visualized in Figure 5.12. My anti-unifier-building tool succeeded in detecting correspondences with special attention to anti-unifying log statements and calculating pairwise similarities in 48 out of 55 test cases. In addition, the instruction coverage of our test cases is 82% as measured with EclEmma.

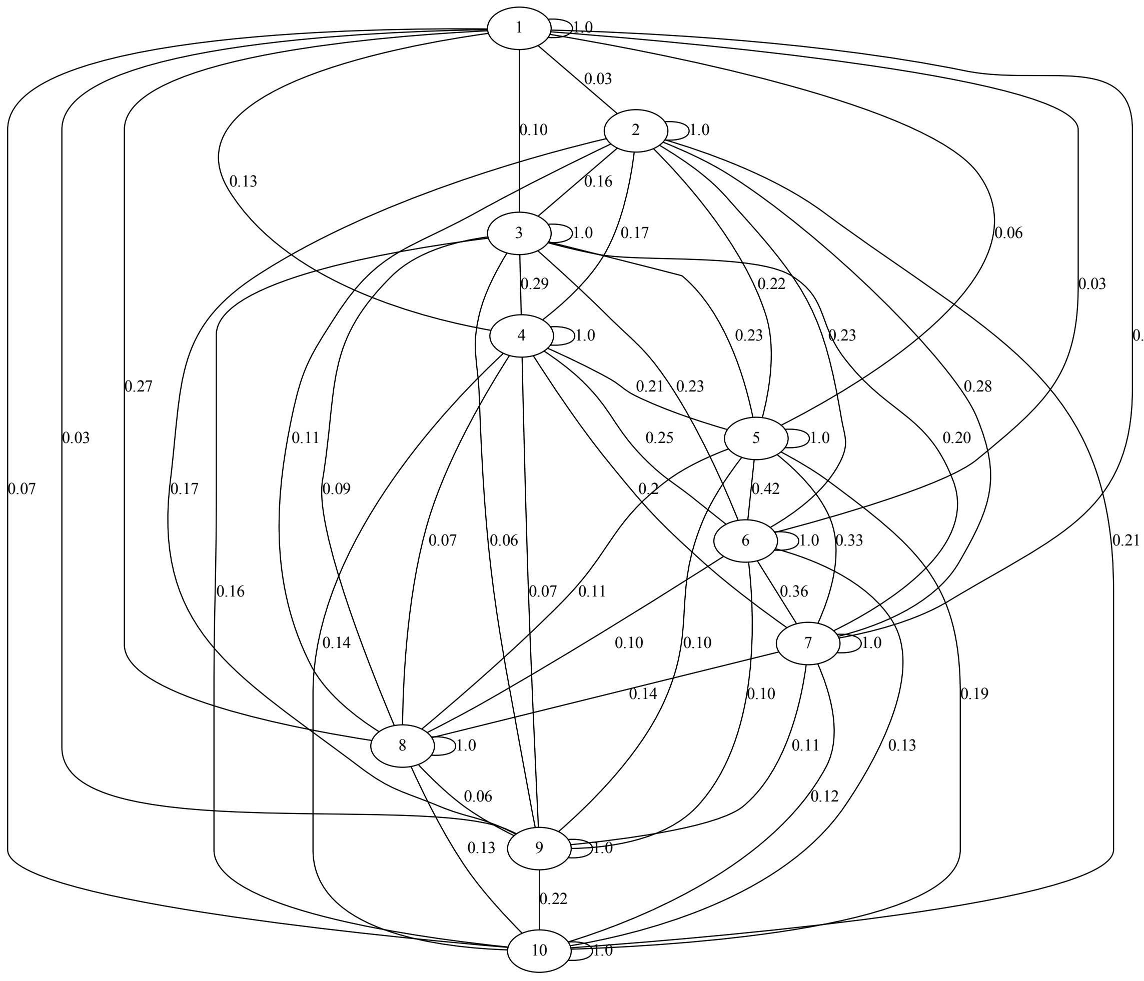


Figure 5.12: A similarity graph representing pairwise similarities calculated by my tool between LMs shown in Table 4.3.

5.6 Summary

I have presented an approach for constructing a generalization from AUASTs of two LMs with special attention to logging statements via structural correspondence. This approach proceeds in three steps. First, several constraints have been applied on the selection of correspondences to prevent anti-unifying log method invocation nodes with any other types of nodes. Second, an approximation technique is employed to find the best correspondence for each AUAST node. Third, the anti-unification of two AUASTs is performed through the application of higher-order modulo theories over the AUAST structures. Furthermore, a measure of similarity has been developed that would provide us with useful information for the clustering phase.

An experimental study was conducted to evaluate the effectiveness of my anti-unification algorithm and the tool support in constructing an anti-unifier from AUASTs of each LM pair of my test suite with a focus on log statements and measuring similarity between them.

Chapter 6

Clustering

In Chapter 5, I proposed an algorithm to construct an anti-unifier from the AUASTs of a pair of LMs, paying special attention to log statements. Recall that the general point of this study is to provide a description of where log statements happen in source code by constructing structural generalizations that represent the detailed commonalities and differences between the AUASTs of LMs. To this end, I require an algorithm that:

- categorizes the methods showing different ways of locating log statements into separate clusters; and
- abstracts the AUASTs of LMs of each group into a structural generalization representing the similarities and differences between them.

In Section 6.1, I describe a modified version of an agglomerative hierarchical clustering algorithm that I developed for my application. The clustering algorithm is a bottom-up approach that starts with singleton clusters, each containing one AUAST, and then it repeatedly merges the closest clusters that are the ones with maximum similarity between their AUASTs. To evaluate my approach, I have implemented the clustering tool and conducted an evaluation through the application of it on the test suite introduced in Section 4.4.1. I describe my empirical study and discuss the results in Section 6.2.

6.1 Modified agglomerative hierarchical clustering algorithm

Clustering is an unsupervised machine learning 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 et al., 1999, Grira et al., 2004]. To perform clustering on a set of AUASTs of LMs, I developed Algorithm 6.1, which is a modified version of the basic agglomerative hierarchical clustering algorithm described in Section 3.7. The clustering algorithm is a bottom-up approach that starts with singleton clusters, each containing one AUAST (line 1), and then it creates a similarity matrix by computing pairwise similarities between cluster pairs (line 2). This step requires defining a notion of cluster similarity. As I aim to construct an anti-unifier for each cluster, the similarity between two clusters is measured based on the similarity between their AUASTs.

In general, this hierarchical algorithm employs an $n \times n$ similarity matrix for a set of n AUASTs, where an element in row i and column j represents the similarity between the ith and the jth clusters. The similarity between two clusters is defined as the similarity between their AUASTs, which is computed through the algorithm described in Section 5.2. However, there are some cases in which the anti-unification of the AUASTs of two clusters does not allow the anti-unification of log statements with one another, since the structures enclosing them are not corresponded. To handle these cases, I adjusted the similarity value between the two clusters to zero. Then the algorithm repeatedly merges the closest clusters that are the ones with maximum similarity between their AUASTs, and updates the similarity matrix by computing the similarity between the new cluster and the old ones (lines 5–6). The merge and update steps are repeated until the similarity between closest clusters becomes below a predetermined threshold value (line 7). Through informal examination, I have found that a threshold value of 0.05 gives the reasonable results, as it allows the classification of methods showing different ways of locating log statements in separate clusters. I also used the anti-unification algorithm described in Section 5.3 to construct an anti-unifier for each cluster.

A hierarchical clustering algorithm can be displayed using a tree-like diagram that shows the order in which the clusters were merged. For example, Figure 6.2 illustrates the modified agglomerative hierarchical clustering process for a sample set of 3 AUASTs using the initial similarity matrix depicted in Figure 6.1. It starts with all AUASTs as singleton clusters. In the first iteration, clusters #1 and #2 are selected as the closest clusters, they are then merged and replaced by

Algorithm 6.1 Modified agglomerative hierarchical clustering algorithm.

```
1: Start with singleton clusters.
2: Compute a similarity matrix.
3: repeat
4:     Find the closest clusters.
5:     Merge the closest cluster pair and replace the original clusters with a new one containing
       the anti-unifier of their AUASTs.
6:     Update the similarity matrix by computing the similarity between new cluster and all re-
       maining clusters.
7: until the similarity between closest clusters becomes below a predetermined threshold value.
```

cluster #4. If the threshold value is set at 0.20, the process should be terminated at this step, as the similarity between the closest clusters (clusters #3 and #4) is below this threshold; otherwise, these clusters should be merged and replaced by cluster #5.

$$similarity = \begin{bmatrix} 1.00 & & \\ 0.48 & 1.00 & \\ 0.12 & 0.17 & 1.0 \end{bmatrix}$$

Figure 6.1: The initial similarity matrix for a sample set of 3 AUASTs.

6.2 Evaluation

To evaluate the clustering approach, I have implemented a tool and conducted an experiment on the set of AUASTs of LMs described in Table 4.3. The clustering tool is an Eclipse plug-in built atop the anti-unifier building tool that inputs a set of AUASTs of LMs extracted from the source code, applies the clustering algorithm on them, and outputs an anti-unifier for each cluster. Then, I have developed some measurements to assess the goodness of the resulting clusters. Recall that clustering should be performed in such a way that the objects within a cluster are similar to one another and dissimilar from the other clusters [Tan et al., 2005]. The measures of cluster evaluation can be divided into two types:

- *Cluster cohesion:* which determines how closely related the objects within a cluster are. In

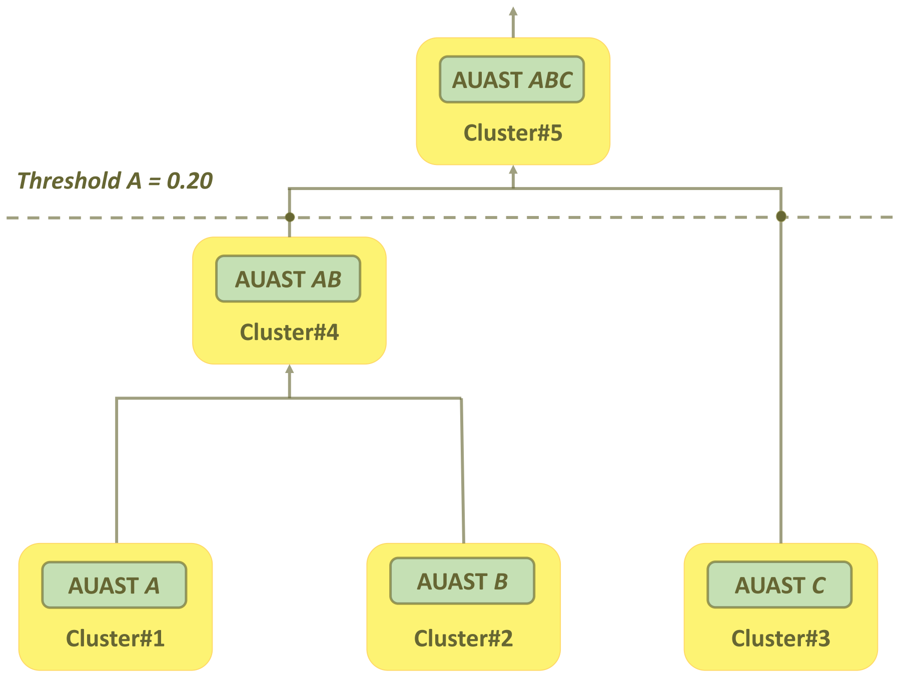


Figure 6.2: Diagram of the modified agglomerative hierarchical clustering process for the sample set.

this experiment, the cohesion of each cluster can be defined as the average of the similarities between the AUAST pairs in each cluster.

- *Cluster separation:* which determines how isolated or well-separated a cluster is from the other clusters. In this experiment, the separation between two clusters can be defined as the similarity between the two cluster anti-unifiers.

6.2.1 Results

I ran the clustering tool on the set of AUASTs of the test suite described in Table 4.3. The cohesion of the clusters produced by applying the clustering tool to the test suite is presented in Table 6.1, where C_i is the ith cluster, and N_i is the cluster size of cluster C_i. In addition, Table 6.2 shows the separateness between each cluster pair. **[RW: What is separateness? The table shows straight zeroes. What does this mean?]**

Cluster	N_i	Cohesion
C_1	4	0.26
C_2	3	0.40
C_3	3	0.32

Table 6.1: The cohesion of clusters.

Cluster	Separateness
C_1–C_2	0
C_1–C_3	0
C_2–C_3	0

Table 6.2: The separateness of clusters.

Clusters C_1, C_2, and C_3 contain logged methods of cases 1, 3, 5, and 8; cases 2, 9, and 10; and cases 4, 6, and 7; respectively. The separation results show that my algorithm was able to generate well-separated clusters. The cohesion results also show that all LMs in the same cluster are related to one another. In general, this experiment shows that my algorithm results in good quality clusters in terms of cohesion and separateness measurements.

6.3 Summary

I have presented a modified version of the agglomerative hierarchical clustering algorithm to categorize logged methods showing different usages of log statements into separate clusters. This algorithm is implemented as an Eclipse plug-in that takes a set of AUASTs of LMs, clusters them into separate groups, and generates an anti-unifier for each cluster. Furthermore, an experimental study was conducted to validate the effectiveness of my clustering algorithm and the tool support on a test suite.

Chapter 7

Characterization Study

To characterize the location of log statements in source code, I conducted an experimental study that addresses the following research questions:

- RQ1: *"Is it possible to find patterns of where log statements occur in source code?"* I aim to investigate whether there are clusters containing a large number of LMs. This suggests that there might be common ways of locating log statements in source code.

- RQ2: *"What common structural characteristics do logged methods have?"* I conducted a manual analysis on the logging usage schemas (LUSs) produced by ELUS to identify the common structural characteristics of LMs in each cluster.

7.1 Experiment

In this experiment, I will analyze logging usage of four popular open-source software systems: Apache Tomcat, Hibernate ORM, Apache Camel, and Apache Solr. Each system is written in the Java programming language and they all utilize the same logging framework, Apache log4j. I decided to study the usage of log4j statements in these systems, as Apache log4j is ranked as the most commonly used logging package for Java[1]. The studied systems are from different application domains: Apache Tomcat is a Java Servlet; Hibernate ORM is an object relational-mapping framework; Apache Camel is a rule-based routing and mediation engine; and Apache Solr is an enterprise search platform. I chose these systems as my study subject due to their popularity in their area of application (7000+ commits to the GitHub repository) and their long history of development (9 to 13 years). Table 7.1 represents the details about these software systems. I also decided

[1] https://en.wikipedia.org/wiki/Java_logging_framework

to exclude the log4j statements at the trace and debug verbosity levels, as they are usually used by developers only during the software development phase. I believe that studying these systems could give us an insight about logging usage in real-world applications.

Software system	**Description**	**Version**	**Start time**	**LOC**	**Log statements**
Tomcat	Server	9.0.11	2003	306,704	3,117
Hibernate ORM	Framework	4.2.23	2004	509,734	1,939
Camel	Middleware	2.18.0	2007	120,528	2,177
Solr	Platform	6.2.1	2007	128,824	2,319

Figure 7.1: Summary of the four software systems used in the characterization study.

My proof-of-concept implementation takes the source code of these systems as inputs, extracts the ASTs of their LMs, applies the proposed algorithm to construct AUASTs, categorizes the AUASTs into clusters, and outputs the structural generalization view for each cluster.

7.1.1 Results

The experimental results for each software system are presented in Table 7.1. This table describes the total number of detected log4j statements (debug- and trace-level log statements are excluded), the number of logged methods (LMs); the number of generated clusters; the number of generalized clusters containing more than one LM; the number of singleton clusters that only contain one LM; and the reduction percentage calculated by the Equation 7.1. In addition, Figure 7.2 shows the histograms of the number of LMs per cluster for each system.

$$reduction = \frac{|Primitive\ clusters| - |Total\ clusters|}{|Primitive\ clusters|} \tag{7.1}$$

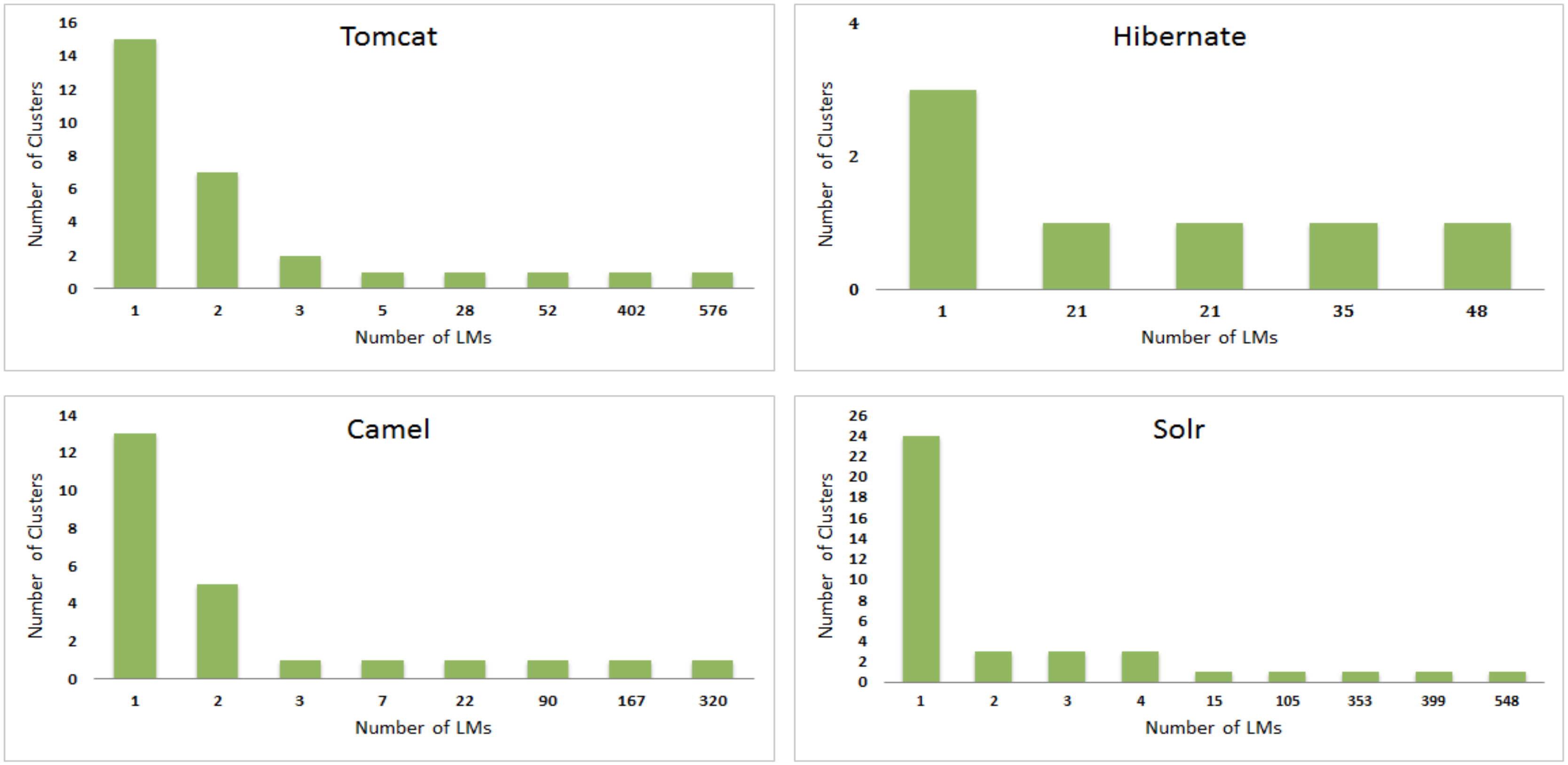


Figure 7.2: Histograms of the number of LMs per cluster.

	Tomcat	Hibernate	Camel	Solr
log4j statements	1098	128	632	1471
LMs	658	81	490	818
Primitive clusters at start	1098	128	632	1471
Non-singleton clusters resulting	14	4	9	14
Singleton clusters resulting	15	3	13	24
Total clusters resulting	29	7	24	38
Reduction	97%	94%	96%	97%

Table 7.1: The experimental results.

7.1.2 Analysis

The first research question is: *"Is it possible to find patterns of where log statements occur in source code?"* As shown in Table 7.1, the number of clusters has been reduced by more than 90% in all the studied systems, indicating that developers follow some patterns for locating the log statements in source code. Furthermore, histograms depicted in Figure 7.2 show that in all the studied systems, a few clusters contain a large number of LMs; however, the other clusters contain a very small number of LMs. This indicates that in these cases, developers follow a more complex or rare way of locating log statements. These exceptions might also happen due to the poor usage of logging statements in source code, which impacts the quality of the entire system negatively.

The second research question is : *"What common structural characteristics do logged methods have?"* To address this question, I manually went through the LUSs to identify the common structural characteristics of locating log statements in source code.

Categorizing logging usage

In this section, I will describe the anti-unifiers of logging usage by examining the LUSs produced by ELUS. In general, there are five main categories of anti-unifiers in the logging usage. Each category represents one cluster of each software system that contains a large number of LMs, that is, the cluster anti-unifier represents a common way of locating log statements in source code. In the following sections, I will describe the common structural characteristics of each category represented by the anti-unifier. In addition, Figure 7.3 presents the number of LMs in each category and its percentage of the total number of LMs for each of the software systems. As shown in this figure, the distribution of logging categories vary from application to application, which implies that logging guidelines should be provided at application-specific level in order to establish effective logging practices.

A. Exception Catch-block Logging

The main common structural characteristics of the anti-unifiers of this category are the **try** statements, where the log statements are located inside the body of a **catch** clause. As shown in Figure 7.3, 14% to 52% of the total LMs are described by the anti-unifiers of this category, and it is the most commonly used logging usage category in the Tomcat and Hibernate software systems. The popularity of this category among all the studied systems is due to the fact that exception handling using the **try/catch** blocks is a common technique in the Java programming language.

B. Conditional Logging

In this category, log statements are enclosed by **if**-statements with their test expressions mostly among *infixExpression*, *methodInvocation*, or *binaryExpression* nodes. The *infixExpression*s mostly either check the equality of an expression to the null literal or tests the validity of the value of a variable; the **if**-statements testing *methodInvocation*s mostly check if the return value of an invoked method is an indicator of a potential problem within a system; and the **if**-statements testing *binaryExpression*s mostly check if a Boolean literal is incorrect. As shown in Figure 7.3, 16% to 37% of the total LMs are described by the anti-unifiers of this category, and it is the most commonly used logging category in the Solr system.

C. Outer Method Logging

In this category, the log statements are located inside the body of *methodDeclaration* nodes but outside of other structures nested therein. A common structural characteristic of the anti-unifiers

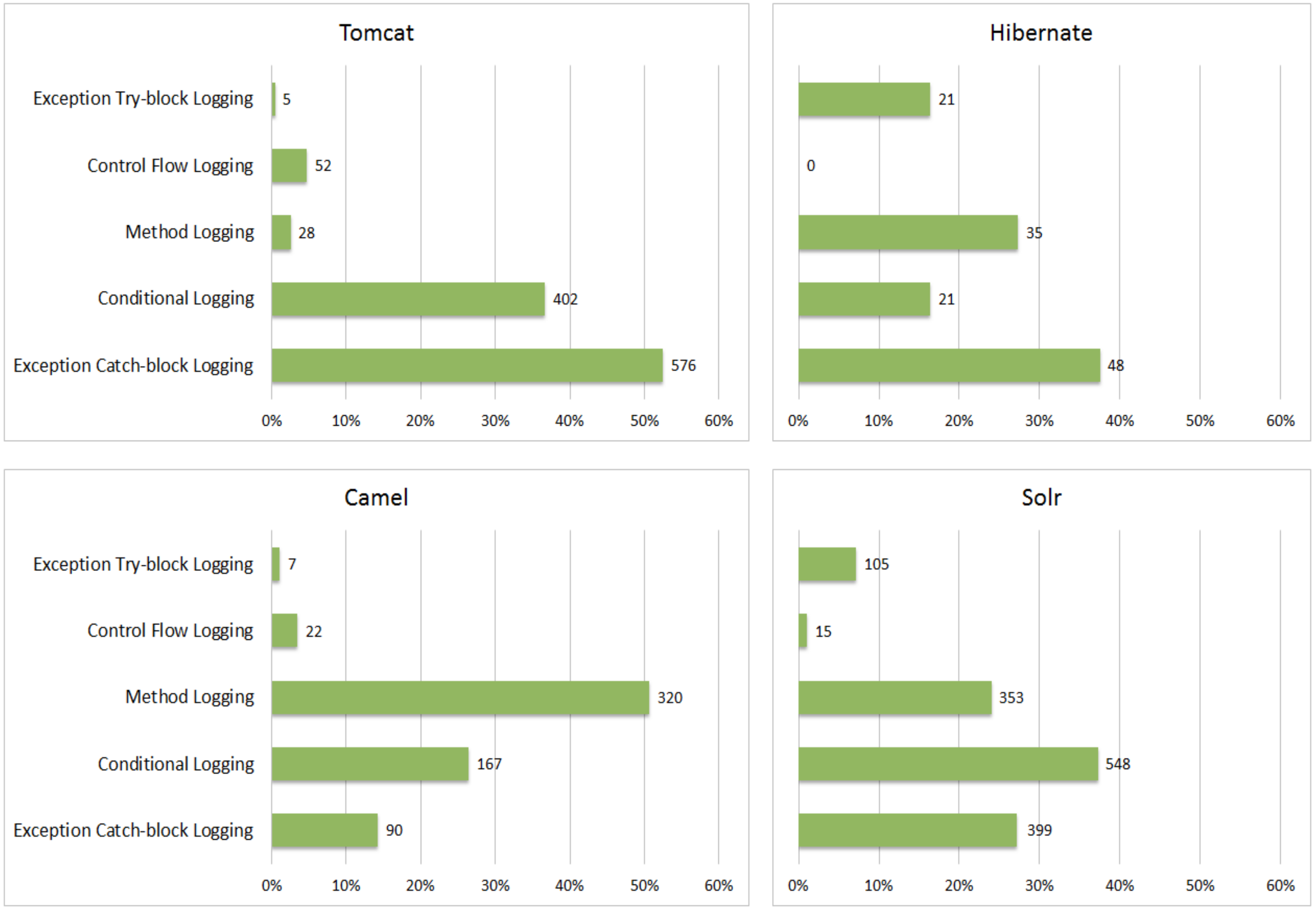


Figure 7.3: The distribution of the categories of anti-unifiers in the logging usage.

in this category is that they mostly use the **throw** statement to throw an exception if an error occurs. The percentage of LMs that are described by the anti-unifiers of this category ranges from 3% to 51%, and it is the most common logging usage category in the Camel software system. This suggests that developers use logging to record important method granularity information about the state of a software system. This information might be used later to detect the root causes of an application problem.

D. Control Flow Logging

In this category, the log statements are located inside the body of either **switch**- or **if-else** statements. These log statements can be used to reveal necessary information to track the location of root causes of a potential problem in a software system. According to the Figure 7.3, 0% to 5% of the total LMs are described by the anti-unifiers of this category.

E. Exception Try-Block Logging

In this category, the log statements are located inside the body of the **try** clause of **try/catch** statements. These log statements can be used to record important information about the code that may throw an exception. As shown in Figure 7.3, 0% to 7% of the total LMs of the studied systems are described by the anti-unifiers of this category.

7.2 Evaluation

An empirical study is conducted to evaluate the quality of the anti-unifiers generated by ELUS in describing the location of log statements in source code. Section 7.2.1 describes the process of evaluating the precision and recall of ELUS.

7.2.1 Calculating the precision and recall

To find the locations in source code that are described by an anti-unifier using ELUS, I applied the DETERMINE-LOCATIONS algorithm, which takes the anti-unifier and a list of all methods in source code and outputs a list of methods that their AUAST matches the anti-unifier AUAST. This algorithm anti-unifies each method in the list with the anti-unifier using the ANTIUNIFY algorithm described in Section 5.3 (lines 2–3). If the result equals the anti-unifier, that method will be added to the list of locations matching the anti-unifier (lines 4–5). EQUALS is a procedure that takes two AUAST nodes and checks whether they are equal or not. To evaluate the generalizability of the anti-unifiers, I have implemented this procedure in two ways: (1) when variables are considered to be *constrained*, it tests that the non-variable nodes are identical in the two AUASTs and checks if the constraints of variable are identical or not; (2) when variables are considered to be *unconstrained*, it tests that the non-variable nodes are identical in the two AUASTs, but permits unconstrained variables to differ. I ran my tool on the source code of the four studied systems and applied this algorithm to find the locations in the code that matches the structure of the generated anti-unifiers. Then, the precision and recall metrics are calculated using Equations 7.2 and 7.3, respectively.

Algorithm 7.1 DETERMINE-LOCATIONS($antiUnifier$,$methods$) finds the locations in source code that matches an anti-unifier.

```
   DETERMINE-LOCATIONS($antiUnifier$,$methods$)
1: $locations \leftarrow \{\}$
2: for $method \in methods$ do
3:     $result \leftarrow$ ANTIUNIFY($antiUnfier, method$)
4:     if EQUALS($result, antiUnifier$) then
5:         APPEND($method, locations$)
6:     end if
7: end for
8: return $locations$
```

$$precision = \frac{TP}{TP + FP} \tag{7.2}$$

$$recall = \frac{TP}{TP + FN} \tag{7.3}$$

Where TP is the number of correct locations obtained, FP is the number of incorrect locations retrieved, and FN is the number of correct locations that were not retrieved. Figures 7.4 and 7.5

show the precision and recall results for each software system where the experiment was run once with constrained variables and once with unconstrained variables.

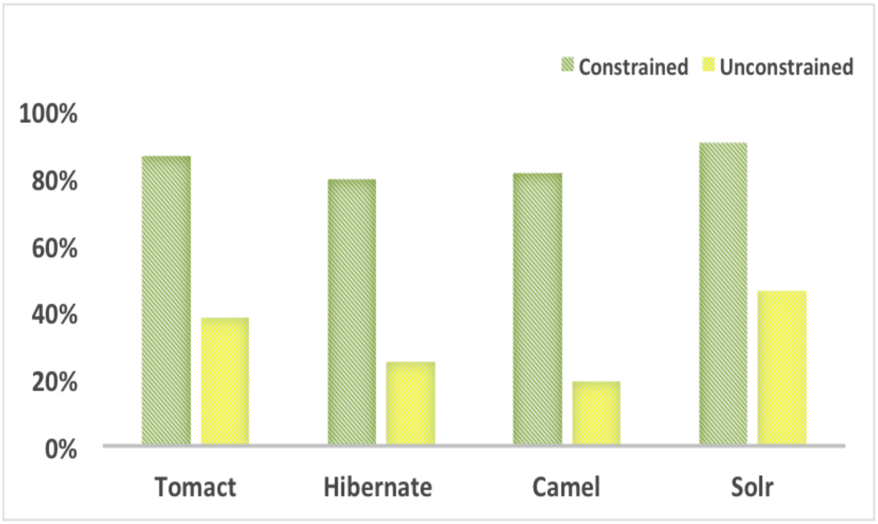


Figure 7.4: The precision of ELUS.

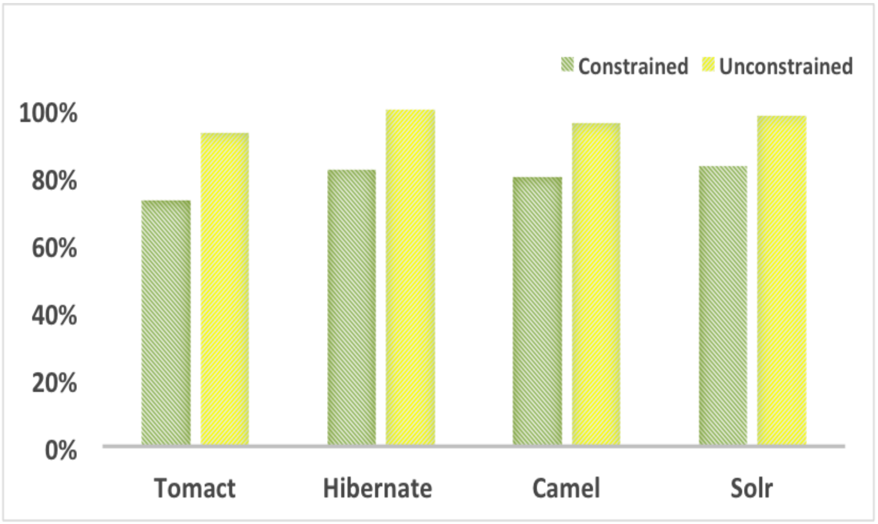


Figure 7.5: The recall of ELUS.

7.2.2 Precision results

The green and yellow bars in Figure 7.4 show the precision results when the experiment was run with constrained and unconstrained variables, respectively. I have also calculated the overall average precision of ELUS, by averaging the precision values between the four software systems.

The average precision for ELUS is 84% and 32% for constrained and unconstrained variable experiments, respectively. In general, the precision for constrained variables is fairly high. The main reason behind the high precision of constrained variables is that in these cases, the variables can only be substituted with some particular nodes, which makes the anti-unifier very specific. However, there are two main reasons for the fact that precision of constrained experiment is not 100%:

1. *Split cases*: To handle the cases containing multiple log statements, I split them into more than one case, where each contains only one logging statement (see Section 5.4.2). However, to find the locations in source code that are described by anti-unifiers using the DETERMINE-LOCATIONS algorithm, I compared them with all the methods in source code without splitting them into multiple cases, which results in retrieving a number of incorrect locations.

2. *Software bugs*: The fact that precision results are not ideal indicates that ELUS has some bugs. In the further work, I aim to improve these results by fixing the software bugs.

According to the Figure 7.4, the precision is fairly low for unconstrained variables. The main reason of the low precision for these cases is the fact that the unconstrained variables can be substituted by any nodes, which makes the anti-unifiers too general. As a result, the tool finds many incorrect locations the matches the anti-unifiers.

7.2.3 Recall results

The green and yellow bars in Figure 7.5 show the recall results when the experiment was run with constrained and unconstrained variables, respectively. I have also calculated the overall average recall of ELUS, by averaging the recall values between the studied systems. The average recall for ELUS is 80% and 97% for the constrained and unconstrained variable experiments, respectively. In general, when variables are constrained, ELUS can detect many correct locations, as the recalls for all the studied systems are fairly high. Also, ELUS can detect most of the correct locations in source code when no constraints are taken on variable nodes.

The main reason behind ELUS's failure to detect the correct locations is the potential complexities in constructing anti-unifiers from a large set of source code fragments. As in some cases, the anti-unifier might not maintain the correct locations of nodes in the AST hierarchy, and thus ELUS would not be able to successfully construct the anti-unifiers of logging usage in source code.

7.3 Usage

The insightful findings of my characterization study regarding the logging usage in several real-world software systems can be used to enhance the quality of existing logging practices by providing some logging guidelines for developers. For example, Figure 7.6 shows a logged method that belongs to a singleton cluster in my experiment. This Java method is an example of a poor usage of a log statement in code, as the list `liveNodes` can be **`null`**, and thus a `NullPointerException` can be thrown causing a system crash. To enhance the quality of the logging usage in this code snippet, a developer my look at our findings to be informed of how usually other developers locate log statements in similar situations. As noted in Section 7.1.2, to avoid the `NullPointerException`, developers usually insert the logging call into the body of an **`if`** statement to check if the value of the variable needed to be logged is not **`null`**. Hence, she can improve the quality of the logging usage in this example by inserting the logging call inside an **`if`** statement and log the needful information if the value of the list `liveNodes` is not **`null`** (lines 6–8 of Figure 7.7). This example demonstrates how these findings can be used in practice to improve the quality of logging practices.

7.4 Summary

I conducted an experimental study to characterize the location of log statements by applying my tool on the source code of four full software systems that make use of the Apache log4j logging framework. My tool inputs the source code of these systems, extracts ASTs of LMs, applies the proposed anti-unification and clustering algorithms, and outputs the anti-unifier for each cluster. I also conducted an experimental study to evaluate the precision and the recall of ELUS in construct-

```
public void setUp() throws Exception {
    SolrZkClient zkClient=new SolrZkClient(zkServer.getZkAddress(),AbstractZkTestCase.
        TIMEOUT);
    for (int i=0; i < 30; i++) {
        List<String> liveNodes=zkClient.getChildren("/live_nodes",null,true);
        Thread.sleep(1000);
        log.info("Waiting for more nodes to come up, now: " + liveNodes.size());
    }
}
```

Figure 7.6: An example of an inappropriate usage of a log statement in a Java method.

```
public void setUp() throws Exception {
    SolrZkClient zkClient=new SolrZkClient(zkServer.getZkAddress(),AbstractZkTestCase.
        TIMEOUT);
    for (int i=0; i < 30; i++) {
        List<String> liveNodes=zkClient.getChildren("/live_nodes",null,true);
        Thread.sleep(1000);
        if (liveNodes != null)
            log.info("Waiting for more nodes to come up, now: " + liveNodes.size());
    }
}
```

Figure 7.7: Modified Java method of Figure 7.6 for the purpose of enhancing the logging usage.

ing the anti-unifiers that describe the location of log statements in source code. This experiment shows that ELUS has achieved promising results in terms of precision and recall. Furthermore, the results taken from the characterization experiment shows that there are common ways of locating log statements. I manually examined the detailed view of structural generalizations and categorized the anti-unifiers of logging usage. In the last section of this chapter, I provided an example to demonstrate the usage of the findings of my characterization study in practice.

Chapter 8

Discussion

In this chapter, I discuss the validity of my evaluation and the characterization study (Section 8.1), and a number of remaining issues including: the limitations and pitfalls of my approach and the tool support (Section 8.2); the usage of anti-unification theory for other applications (Section 8.3); and the definition of an intermediate form of structural variable constraints (Section 8.4).

8.1 Threats to validity

8.1.1 Threats to internal validity

- *Researcher bias:* The results of my manual analysis might be biased, as I determined the correct correspondences for my test suite myself. To limit the bias, third-parties could be involved to double-check the accuracy of my manual inspection in future work.

- *Limited sample size:* Another potential threat is that the rate of detecting correct best correspondences by my tool might have happened coincidentally for my test suite, as the test suite size is limited in my experiments. To reduce this doubt, I manually examined the cases where my tool fails to detect correct correspondences, and I found that the failures are due to the fundamental limitations and complexities in the construction of structural generalizations through the use of structural correspondence. That is, my tool creates structural generalization successfully with regard to what my algorithm should generate.

8.1.2 Threats to external validity

- *Representativeness of the studied systems:* A potential threat to the validity of the exploratory study is the degree to which my selected set of software systems is representative of all real-world logging practices. To address this issue, I selected various open-source software projects in terms of application domain. The studied software systems have been widely used by many developers for a long period of time. However, the fact that I only studied Apache log4j statements might limit the generalizability of the findings. To improve the generalizability of this study, I have chosen one system not from the Apache Software Foundation. However, my findings might reflect the characteristics of logging usage in systems that are closed source, from other domains, or written in programming languages other than Java. In addition, the experiments have examined one set of logged methods (LMs) from a real-world software system; different sets and different systems may generate different results. Despite the fact that I cannot claim that the set of tested LMs is representative of all LMs in real-world software systems, the experimental results are still promising, as the location of logging statements in the tested methods differ markedly from each other with respect to their location and details. Hence, these experiments have sufficed to indicate the effectiveness of my approach in constructing structural generalizations.

- *The number of software systems under study:* My study was conducted on four large open-source software systems written in Java. While the studied systems are limited in number, I believe that my findings of the logging practices in these systems could be generalizable to many other software systems. This threat could be reduced by more experiments on wider range of software systems in future work.

- *The number of researchers who performed the manual examination:* Due to the limitations during the course of my research, the manual examination was conducted only by the first author. Future work can reduce this thread by involving other people to verify the results.

8.1.3 Threats to construct validity

As there is no existing benchmark dataset to assess the quality attributes of the anti-unifiers in terms of accuracy, reliability, and performance, I have applied the DETERMINE-LOCATIONS algorithm (Algorithm 7.1) to measure the performance of ELUS in terms of exactness (precision) and completeness (recall) for the studied systems. However, I acknowledge that these metrics may not perfectly measure all the quality attributes of the anti-unifiers of logging code. Another potential construct threat is the degree to which my similarity metric perfectly represents how similar logged methods are. To reduce this threat, I conducted an experimental study in which I manually estimated the similarity between the usages of logging statements in different methods of a sample test suite and compared the similarity values with the results extracted by the automated approach.

8.2 The pitfalls of my tool

There are some issues that the approximation approach and my tool support are not able to handle perfectly, including inaccurate node ordering, and the resolution of conflicts that happened in constructing the anti-unifiers.

8.2.1 Inaccurate node ordering

My anti-unification algorithm does not guarantee to maintain the correct sequence of statements in the body of methods in the case of anti-unifying two method declaration nodes, as the order of statement nodes is not considered in determining the best correspondences. For example, consider that we have two corresponding methods $method_1$ and $method_2$ involved in two sequences of statements $\{a_1, a_2\}$ and $\{b_1, b_2\}$. If the b_1 and b_2 nodes are found to be the best correspondences of the a_2 and a_1 nodes, respectively, a_1 will be anti-unified with b_2 and a_2 will be anti-unified with b_1 to construct the structural generalization. Therefore, the anti-unification algorithm does not preserve the correct ordering of nodes in the original structures.

8.2.2 Conflict resolution

The decisions that I made, to resolve these conflicts, occurred in constructing structural generalizations that might affect the accuracy of our results. For example, in situations where I have two correspondences with the same similarity value in the ordered list of correspondence connections, my approach picks the one which involves two subtrees with the higher number of nodes, though this might not be the best choice for all cases. In addition, I consider AST hierarchies to perform anti-unification. That is, my algorithm does not anti-unify two nodes if their parent nodes are not found to also correspond. As a result, situations can occur where in fact two nodes should be anti-unified with each other, while they are not anti-unified by the tool. Though these decisions led to results that represent suboptimal anti-unifiers, they helped to limit the complexity of my approach, allowing the implementation of it to be a practical solution.

8.3 Applications of anti-unification

My study demonstrates the application of an extended from of anti-unification (higher-order anti-unification modulo theories) to infer usage patterns of log statements in source code via the creation of structural generalizations. Anti-unification and its extensions have previously been applied to solve several theoretical and practical problems, such as analogy making [Guhe et al., 2010], determining lemma generation in equational inductive proofs [Burghardt, 2005], and detecting the construction laws for a sequence of structures [Burghardt, 2005].

Higher-order anti-unification modulo theories (HOAUMT) can be used to create generalizations in different contexts, and therefore the set of equational theories must be developed specifically for the higher-order structure found in each problem context. That is, the utility of these theories are highly dependent on how well they allow the incorporation of semantic knowledge of structures. In addition, useful theories should ensure that only a finite number of instances exist for each structure. The practical experiments I have conducted through the application of my tool on a test suite demonstrate that an approximation of HOAUMT can be successfully used to construct structural generalizations required to solve a problem.

8.4 Defining intermediate constrained variables

As explained in Section 7.2, I conducted an empirical experiment to evaluate the quality of the anti-unifiers for both constrained and unconstrained variables. As shown in Figures 7.4 and 7.5, the average precision and recall values for the unconstrained experiment is lower than and higher than, respectively, the average precision and recall values for the constrained experiment. However, to keep the balance between the exactness (precision) and completeness (recall) of the anti-unifiers, an intermediate form of constraints can be defined for structural variables. That is, instead of constraining to only the specific substitutions, intermediate constraints could be defined such that only specific types of nodes (e.g., *infixExpression*) could be used to substitute a structural variable. A future experiment can be run to see whether considering variables with intermediate constraints would yield better overall results than the fully constrained and unconstrained forms.

8.5 Summary

I discussed the potential threats to validity of my evaluation and characterization study. To limit the bias of my experiments, I selected the test cases form a real system with various levels of similarity in the usage of log statements. Furthermore, I examined the failed test cases to ensure that my tool works when it should work with regard to the proposed algorithm. I will also make my test suite available for public examination to further check the accuracy of my manual inspection. For the characterization study, I selected various software systems in terms of functionality. I also discussed the remaining issues with the tool support, including inaccurate node ordering and handling the conflicts happened in the construction of anti-unifiers.

This work aims to provide a detailed view of structural generalizations constructed from a set of source code fragments that use log statements via the application of anti-unification and clustering. However, I argued how higher-order anti-unification modulo theories can be effectively approximated for various applications by means of developing an appropriate set of equational theories particularly for the higher-order structure used in each problem context. I also explained how the definition of a form of intermediate constraints on structural variables may lead to better experimental results.

Chapter 9

Related Work

In this chapter, I review related work to the topics of my study including: the application of logging in real-world software systems (Section 9.1), understanding the existing logging practices (Section 9.2), determining correspondences in source code (Section 9.3), data mining approaches to extract API usage patterns (Section 9.4), and anti-unification and its application to detect structural correspondences and construct generalizations (Section 9.5).

9.1 Usage of logging

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

- *Log analysis for failure diagnosis:* Xu et al. [2009] use statistical techniques to learn a decision tree based signature from console logs and then utilize the signature to diagnose anomalies. SherLog [Yuan et al., 2010] uses failure log messages to infer the source code paths that might have been executed during a failure. Jiang et al. [2009] 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.
- *Log analysis for system behaviour understanding:* Fu et al. [2013] present an approach for understanding system behaviour 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 cause 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 a 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 behaviours extracted from the log data.

9.2 Understanding existing logging practices

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. [2012b] provides a quantitative characteristic study to investigate log message modifications of four open-source software systems by mining their revision history. They have also discovered that developers are continuously making an effort to modify the context of log statements to improve the quality of logging practices. Shang et al. [2015] performed an empirical study to find the relation between logging characteristics and software quality. They found that log-related metrics are good predictors for post-release defects.

Kabinna et al. [2016b] empirically studied the stability of log statements in four open source software systems. They model the historical log statement changes using a data mining classifier. They find that developer experience, file ownership, log density and SLOC are important indicators of whether a log statement will change in the future or not.

Kabinna et al. [2016a] studied the logging library migration of several software systems within the Apache Software Foundation. They found that the migration of logging libraries happened in these systems in order to enhance the system's flexibility and performance, and also to utilize more advanced functionalities. Furthermore, they discovered that migration is not a trivial task, as 70% of the migrated projects encounter post-migration bugs.

Yuan et al. [2012a] developed Errlog, a tool that automatically inserts a log statement in source code when a generic error condition happens, in order to assist failure diagnosis. LogEnhancer [Yuan et al., 2012c] automatically enhances existing log messages by detecting variables containing important values and inserting them into the log messages. However, these studies only consider source code fragments containing bugs that are needed to be logged and do not consider the other code fragments with no bugs that were still logged. Moreover, these studies mainly research log message modifications and potential enhancements of them; in contrast, the focus of my study is on understanding where log statements are used in source code.

Fu et al. [2014] and Zhu et al. [2015] applied a data mining approach to detect the main factors impacting the location of logging statements. Fu et al. [2014] also conducted a survey to validate their findings. Ding et al. [2015] proposed a constraint-based approach to decide whether to log for each logging request at run-time in order to minimize the performance overhead. In contrast, my study is the first work that automatically characterized logging usage in source code by constructing anti-unifiers that represent the structural similarities and differences among a set of source code fragments containing logging statements.

9.3 Correspondence

Several studies have been conducted to find the similarities and differences between 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 and then finds 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 source code. Holmes et al. [2006] recommend 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 code in the repository. Coogle [Sager et al., 2006] was developed to detect similar Java classes by converting ASTs to a normalized format and then comparing them through tree similarity algorithms. However, none of these approaches construct a detailed view of structural generalizations needed in my context.

Cossette et al. [2014] present a new approach, called matching via structural generalization (MSG), to recommend replacements for API migration. They applied Jigsaw to find structural correspondences; however, their algorithm does not suffice to construct structural generalizations that represent the detailed commonalities and differences of a set of source code fragments with special attention to log statements, which is required to solve my problem.

9.4 API usage patterns

Various data mining approaches have been used to extract API usage patterns out of 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 source code.

Another 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, by calculating popularity statistics for each API of a library. Acharya et al. [2007] present a framework to extract API usage scenarios as partial orders, as 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 construct the detailed structural generalizations needed in my context.

9.5 Anti-unification

Anti-unification is the problem of finding the most specific generalization of two terms. First-order syntactical anti-unification was introduced by Plotkin [1970] and Reynolds [1970], independently. Burghardt and Heinz [1996] extend the notion of anti-unification to E-anti-unification to incorporate background knowledge to syntactical anti-unification, which is required for some applications. Anti-unification and its extensions have been applied in various studies for program analysis. Bulychev and Minea [2009] suggest an anti-unification algorithm to detect clones in ASTs. Their approach consists of three stages: first, identifying similar statements through anti-unification and grouping them into clusters; second, determining similar sequences of statements with the same cluster identifier; third, refining candidate statement sequences using an anti-unification based similarity measurement to generate final clones. However, their approach does not construct structural generalizations.

Cottrell et al. [2007] propose Breakaway to automatically determine structural correspondences between a pair of ASTs to create a generalized correspondence view. However, their approach does not allow the determination of the best structural correspondence for each AST node required to my context, and only approximates higher-order AU (not HOAUMT). Cottrell et al. [2008] extended their earlier work to HOAUMT in order to develop Jigsaw, a tool to help developers integrate small-scale reused code into their own source code by determining structural correspondences. Although I used the Jigsaw framework to find potential correspondences between AST nodes, their approach does not suffice to construct structural generalizations from a set of source code fragments by considering the limitations of this study in determining correspondences.

9.6 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 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. Several data mining and statistical source code analysis techniques have been used to extract API usage patterns, however, none of them enable us to construct the detailed structural generalizations of a set of source code fragments. On the other hand, using higher-order anti-unification modulo theories and an agglomerative hierarchical clustering algorithm allow us to construct generalizations representing the commonalities and differences between ASTs of logged methods and grouping them into clusters based on the structural correspondences.

Chapter 10

Conclusion

Logging is a common programming practice to gain valuable information about the execution of a software system. In practice, effective usage of log statements in source code is needed to record important run-time information without causing unintentional consequences (e.g., performance overhead). However, it is a challenging task to write high-quality logging code as the current logging practices are not well-supported, and developers are not provided with enough guidance on how to make effective logging decisions. In this study, I proposed an approach that automatically characterize the location of log statements in source code from the point of view of methods containing them (logged methods). My approach aims to construct structural generalizations that describe the structural similarities and differences between logged methods.

I have developed a prototype tool, called ELUS, to implement the proposed approach that proceeds in four steps. First, it extracts the ASTs of logged methods using the Eclipse JDT framework, extends the AST structures to AUAST, and determines potential structural correspondences via the Jigsaw framework. Second, it constructs an anti-unifier from the AUASTs of two given logged methods with a focus on log statements through the application of higher-order anti-unification modulo theories. Due to the problem of undecidability of HOAUMT, it employs an approximation technique which greedily determines the best correspondence for each node with the highest similarity. It applies several constraints prior to determining the best correspondences to prevent the anti-unification of log statements with any other types of nodes. It also uses a measure of structural similarity that determines how similar is the usage of logging statements in different methods. Third, it categorizes a set of logged methods via a hierarchical clustering algorithm suited to my application. Fourth, it generates a detailed view of the anti-unifier constructed from each cluster to describe the structural similarities and differences between logged methods of the cluster.

To evaluate the effectiveness of this approach in constructing generalizations and clustering logged methods, three experiments were conducted on a sample test suite. I found that my tool was successful in determining correct correspondences in 87% of test cases. It was also successful in creating well-separated clusters of logged methods of my test suite. This work also shows how the Jigsaw framework could be effectively used to construct structural generalizations for a particular problem context by determining structural correspondences. To characterize the location of log statements in source code, I applied my tool on the source code of four software systems from various application domains: Tomcat, Hibernate, Camel, and Solr. My characterization study resulted in five main categories of locations of log statements in source code. Furthermore, an empirical experiment has been conducted to evaluate the performance of ELUS. This experiment shows that ELUS has an average precision of 84% and recall of 80% for the studied software systems.

In summary, my study makes the following main contributions:

- I proposed a novel approach to automatically characterize logging usage in source code, inspired by anti-unification and clustering.
- I implemented the proposed approach as an Eclipse plug-in tool, named ELUS, and conducted a within-project evaluation using four large open-source software systems.
- I conducted an exploratory experimental study by applying ELUS on the source code four large open-source software systems.
- My analysis has found the feasibility of finding common patterns of where log statements occur in source code.
- My analysis has resulted in five main categories of logging usage in source code, including exception catch-block logging, conditional logging, outer method logging, control flow logging, and exception try-block logging.

- My analysis showed that the distribution of logging categories vary from application to application.
- I illustrated the potential applicability of my findings to enhance existing logging practices using an example.

10.1 Future Work

Future work could be directed to address the remaining issues of this study as described in the following sections.

10.1.1 Improving logging practices

Characterizing logging usage could be used to improve logging practices through the provision of policies and guidelines that might help developers in making informed decisions about where and what to log. Further studies could be conducted to investigate the feasibility of predicting the location of log statements based on the detected usage patterns. Future work can also be done to develop recommendation tool support that not only saves developers' time and effort for making decisions about where and what to log, but also improve the quality of logging practices.

10.1.2 Further extensions to my approach and the tool support

In the future, I aim to improve the precision and recall of ELUS by repairing software bugs and by either (1) altering the way in which I calculate precision and recall or (2) replacing my approach for splitting complex cases. Additional work can also be done to improve the accuracy of my approach and the tool support by incorporating additional data flow analysis and natural language processing techniques. The data flow analysis can be performed to detect the problems related to node ordering in the construction of anti-unifiers. My approach can be extended to examine more advanced semantical and contextual information of source code fragments enclosing log statements in addition to structural information. Furthermore, further analyses can be done to detect and resolve all the conflicts happen in deciding the best correspondences to construct an approximation of the best anti-unifier to my problem. However, the complexity of applying all these extensions must be controlled to maintain the approach as a practical one.

10.1.3 Further validation of this study

The characterization study can be conducted on more software systems to further validate the findings of this study. In addition, a survey can be performed to gain more feedback from developers to investigate the factors they consider when they want to decide on the location of log statements. It might also be helpful to recognize important structural and semantic information that should be taken into account for characterizing logging usage.

10.1.4 Other applications

Any applications that are involved in the inference of structural patterns in source code even infrequently-used patterns might benefit from my tool's underlying framework. Furthermore, understanding the commonalities and differences amongst source code fragments has application in several areas of software engineering, such as API usage pattern summarization, code clone detection, recommendation of replacements for API migration, and merge of different branches in a version control system. My tool's functionality to construct the detailed view of structural generalizations from a set of source code fragments could be used to improve the results of such studies as well.

Bibliography

Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 25–34, 2007.

Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Data Bases*, volume 1215, pages 487–499, 1994.

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 2–13, 2004.

Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 368–377, 1998.

Doug Beeferman and Adam Berger. Agglomerative clustering of a search engine query log. pages 407–416, 2000.

Matt Bishop. A model of security monitoring. In *Proceedings of the IEEE Annual Computer Security Applications Conference*, pages 46–52, 1989.

Peter Bulychev and Marius Minea. An evaluation of duplicate code detection using anti-unification. In *Proceedings of the International Workshop on Software Clones*, 2009. URL http://www.informatik.uni-bremen.de/st/IWSC/bulychev.pdf.

J. Burghardt. E-generalization using grammars. *Artificial Intelligence Journal*, 165(1):1–35, June 2005. doi: 10.1016/j.artint.2005.01.008.

Jochen Burghardt and Birgit Heinz. *Implementing Anti-unification Modulo Equational Theory*. GMD-Forschungszentrum Informationstechnik, 1996.

Bryan M. Cantrill, Michael W. Shapiro, Adam H. Leventhal, et al. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 15–28, 2004.

Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. The dimension of separating requirements concerns for the duration of the development lifecycle. In *Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems*, 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. Technical Report 2014-745-10, Department of Computer Science, University of Calgary, Calgary, Canada, May 2014.

Rylan Cottrell, Joseph J. C. Chang, Robert J. Walker, and Jörg Denzinger. Determining detailed structural correspondence for generalization tasks. In *Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 165–174, 2007. doi: 10.1145/1287624.1287649.

Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 214–225, 2008. doi: 10.1145/1453101.1453130.

William H. E. Day and Herbert Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1(1):7–24, 1984.

Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *Proceedings of the USENIX Annual Technical Conference*, pages 139–150, 2015.

Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the IEEE International Conference on Data Mining*, 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 International Working Conference on Mining Software Repositories*, pages 397–400, 2013.

Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the International Conference on Software Engineering*, pages 24–33, 2014.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Addison-Wesley, Java SE 7 edition, 2012. URL http://docs.oracle.com/javase/specs/jls/se7/html/index.html.

Nizar Grira, Michel Crucianu, and Nozha Boujemaa. Unsupervised and semi-supervised clustering: a brief survey. *A review of machine learning techniques for processing multimedia content*, 1:9–16, 2004.

Markus Guhe, Alison Pease, Alan Smaill, Martin Schmidt, Helmar Gust, Kai-Uwe Kühnberger, and Ulf Krumnack. Mathematical reasoning with higher-order anti-unifcation. In *Proceedings of the 32nd Annual Conference of the Cognitive Science Society*, pages 1992–1997, 2010.

Samudra Gupta. *Pro Apache Log4J: Java Application Logging Using the Open Source Apache Log4J API*. Apress, 2 edition, 2005. doi: 10.1007/978-1-4302-0034-5.

Reid Holmes and Robert J Walker. Informing eclipse api production and consumption. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 70–74. ACM, 2007.

Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32 (12):952–970, December 2006. doi: 10.1109/TSE.2006.117.

A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31 (3):264–323, September 1999. doi: 10.1145/331499.331504.

Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding customer problem troubleshooting from storage system logs. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 43–56, 2009.

Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. Logging library migrations: A case study for the Apache Software Foundation projects. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 154–164, 2016a.

Suhas Kabinna, Weiyi Shang, Cor-Paul Bezemer, and Ahmed E. Hassan. Examining the stability of logging statements. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 326–337, 2016b.

George Karypis, Eui-Hong Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. *SIGSOFT Software Engineering Notes*, 30(5): 306–315, 2005.

Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *Proceedings of the USENIX Annual Technical Conference*, pages 24:1–24:14, 2010.

Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 167–176, 2000. doi: 10.1109/ICSE.2000.870408.

Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 353–366, 2012.

Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Toward models for forensic analysis. In *Proceedings of the International Workshop on Systematic Approaches to Digital Forensic Engineering*, pages 3–15, 2007.

Gordon D. Plotkin. A note on inductive generalization. *Machine intelligence*, 5(1):153–163, 1970.

Edie Rasmussen. Clustering algorithms. In William B. Frakes and Ricardo Baeza-Yates, editors, *Information Retrieval*, pages 419–442. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-463837-9. URL http://dl.acm.org/citation.cfm?id=129687.129703.

John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine intelligence*, 5(1):135–151, 1970.

Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5): 613–637, 2013.

Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar Java classes using tree algorithms. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 65–71, 2006.

Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27, 2015.

Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*, chapter 8. Cluster Analysis: Basic Concepts and Algorithms. Pearson, 2005. ISBN 978-0321321367.

Ellen M. Voorhees. Implementing agglomerative hierarchic clustering algorithms for use in document retrieval. *Information Processing & Management*, 22(6):465–476, 1986.

Tao Xie and Jian Pei. MAPO: Mining API usages from open source repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 54–57, 2006. doi: 10.1145/1137983.1137997.

Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, pages 117–132, 2009.

Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the USENIX Annual Technical Conference*, pages 2:1–2:14, 2000.

Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. *ACM SIGARCH Computer Architecture News*, 38(1):143–154, 2010.

Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 293–306, 2012a.

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

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

Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, volume 1, pages 415–425, 2015.