UNIVERSITY OF CALGARY

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

August, 2016

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

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

Dr. Robert J. Walker
Supervisor
Department of Computer Science

Dr. Jörg Denzinger
Examiner
Department of Computer Science

Dr. Christian J Jacob
Examiner
Department of Computer Science

Date

Abstract

Understanding the similarities and differences between a set of source code fragments has many applications in various software engineering research areas. As a specific application, it can be employed to study where developers log in practice. Logging tasks are mostly performed by developers to monitor, troubleshoot or debug a software system. Several logging frameworks have been especially created to help developers perform logging practices, but they do not support locating log statements in source code. This leaves developers with the burden of deciding where to log manually. If logging is properly done, it can provide valuable information for software development and maintenance. On the other hand, the costs of logging inappropriately may outweigh its benefits. So far, few studies have been conducted to characterize logging usage in real-world applications. This work presents an automated approach to characterize the location of log statements in source code through the approximation of a generalization technique (higher-order anti-unification modulo theories) and through a hierarchical clustering approach to construct a set of logging usage schemas, each describes the commonalities and differences amongst source code fragments containing log statements This approach has been refined in a prototype tool, called ELUS, that greedily identifies the best structural correspondences with respect to the highest similarity and some constraints. A characterization study has been conducted through the application of the tool on the source code of five software systems to perform a per-system method-granularity analysis. Our experimental results show that . Two empirical evaluations were conducted in this study: (1) an experiment was conducted to validate the effectiveness of the proposed approach through the application of its supporting tool on a test suite. (2) an empirical experiment has been performed to assess the accuracy of the characterization study.

Acknowledgements

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

AST	abstract syntax tree
AU	anti-unification
AUAST	anti-unification abstract syntax tree
HOAUMT	higher-order anti-unification modulo theories
LM	logged method

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 been usually 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 to log suggests that in practice effective logging is not a straightforward task to perform. 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 on 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 to perform. 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 logs are used in source code. The context of log statements is important to perform high quality logging, as it provides necessary runtime information needed for system analysis. The location of log statements has also 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 for log system analysis. 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 log statements [Yuan et al., 2012c, 2010]. Yuan et al. [2012a] proposed Errlog to automatically inserts additional log statements into a software system to log all the generic exceptions 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 in where log statements do occur in source code through the construction of a detailed view of structural generalizations that describe the commonalities and differences between source code fragments containing log 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 to describe the logged event and some optional variables related to the event. The verbosity level is intended to classify the severity of a logged event such as a debugging note, a minor issue, or a fatal error. Figure 1.1 provides examples of logging calls from the 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 concise description of where logs are used in source code by constructing generalizations that represent the detailed 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 implement 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 detailed 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 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. To evaluate the approach, I implemented it in a tool called ELUS, written in the Java programming language. I use the Eclipse JDT framework to extract the AST of LMs from a Java program, and employ 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 (Section 5.4), and my clustering tool is developed atop of it to perform the clustering algorithm described in Section 6.1.

1.3 Thesis statement

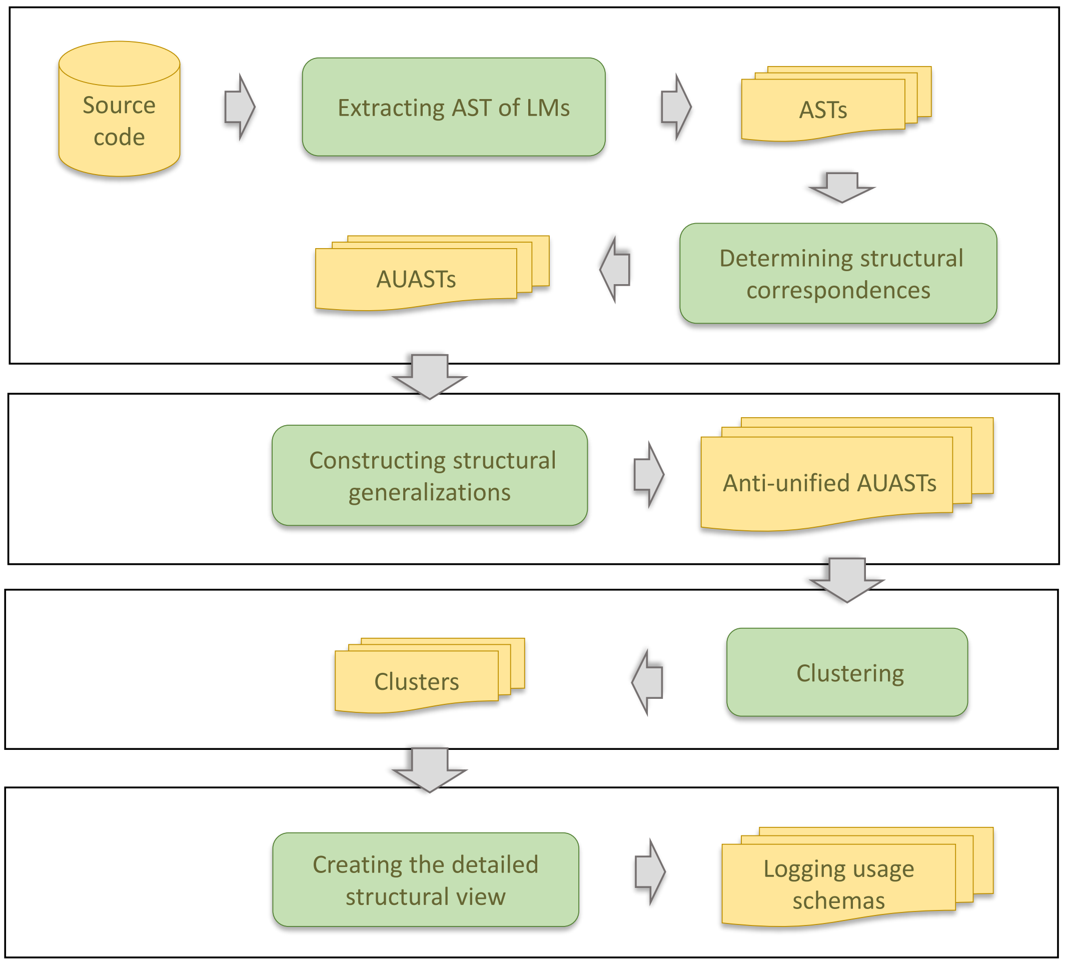
The thesis of this work is to characterize where log statements do occur in source code by constructing structural generalizations describing the commonalities and differences between source code fragments that contains log statements, thus providing the developers with an insight on where to log.

1.4 Thesis organization

The remainder of the thesis is organized as follows.

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 insufficiently 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



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

Figure 1.2: Overview of the approach.

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 through the application of 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 in five 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 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 logging quality suggests that using print statements is not sufficient for real-world applications.

The logging framework offers many more features that are not possible using print statements. In most logging frameworks (e.g., log4j, SLF4j, java.util.logging), different verbosity levels of logging are available for use. That is, by logging at a particular log level, messages will get logged for that level and all levels above it and not for the levels below. As an example, debug log level messages can be used in a test environment, while 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. In addition, most logging frameworks allow the production of formatted log messages, which makes it easier for a developer to monitor the behaviour of a system. Furthermore, when one is working on a server side application, the only way to know what is going on inside the server is by monitoring the log file. 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 in great detail in order to perform it 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 this system (Figure 2.1) using the log4j logging 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 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 logging calls. She thinks that it is important to log the information related to a message sent to a registered component, including the message content and the transmission time, to find the root causes of potential errors in sending messages. She simply wants to begin by using a logging call at the start of the send() method (line 2 of Figure 2.2) to log the information. However, she realizes that this logging call does not allow her to log the information she wants, as the time variable is not initialized at the beginning of this method. Therefore, she proceeds to examine the body of the send() method line-by-line and uses another logging call after the time variable is initialized inside an **if** statement that checks that the value of the variable time is not invalid (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 ex-

```
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.

```
public static void send(EBMessage message){
    //logging call
    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 logging calls for the send(EBMessage) method.

```
public static void send(EBMessage message) {
    // logging call
    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 != 0){
            // logging call
        }
    }
}
```

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

ceptions in the Java programming language. She creates a **try/catch** block to capture the potential failure in sending messages, and uses a logging call inside the **catch** block to log the exception (shown in lines 2–16 of Figure 2.4). However, she realizes that using this logging call 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 to use logging calls to perform the logging task of the EditBus class. By making appropriate decisions about where to use logging calls, 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 logging calls 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 logging calls more quickly, and so she could pay more attention to the remaining, conceptually complex issues to complete the logging task.

```
public static void send(EBMessage message){
    try {
        // logging call
        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 != 0){
                // logging call
            }
        }
    } catch(Throwable t) {
        // logging call
    }
}
```

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

```
public static void send(EBMessage message) {
    // logging call
    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 != 0) {
                // logging call
            }
        } catch(Throwable t) {
            // logging call
        }
    }
}
```

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

```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) {
        // logging call
        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 != 0) {
                    // logging call
                }
            } catch(Throwable t) {
                // logging call
            }
        }
    }
}
```

Figure 2.6: The developer's final determination of the usage of logging calls 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 we represent and analyze source code, we 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 we 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 3.5.

3.1 Concrete syntax trees and abstract syntax trees

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

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

Figure 3.1: A simple example Java program.

represents; leaf nodes represent the concrete lexemes, such as literals and keywords.

We focus on the Java programming language and we 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, we 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; we 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, although we 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

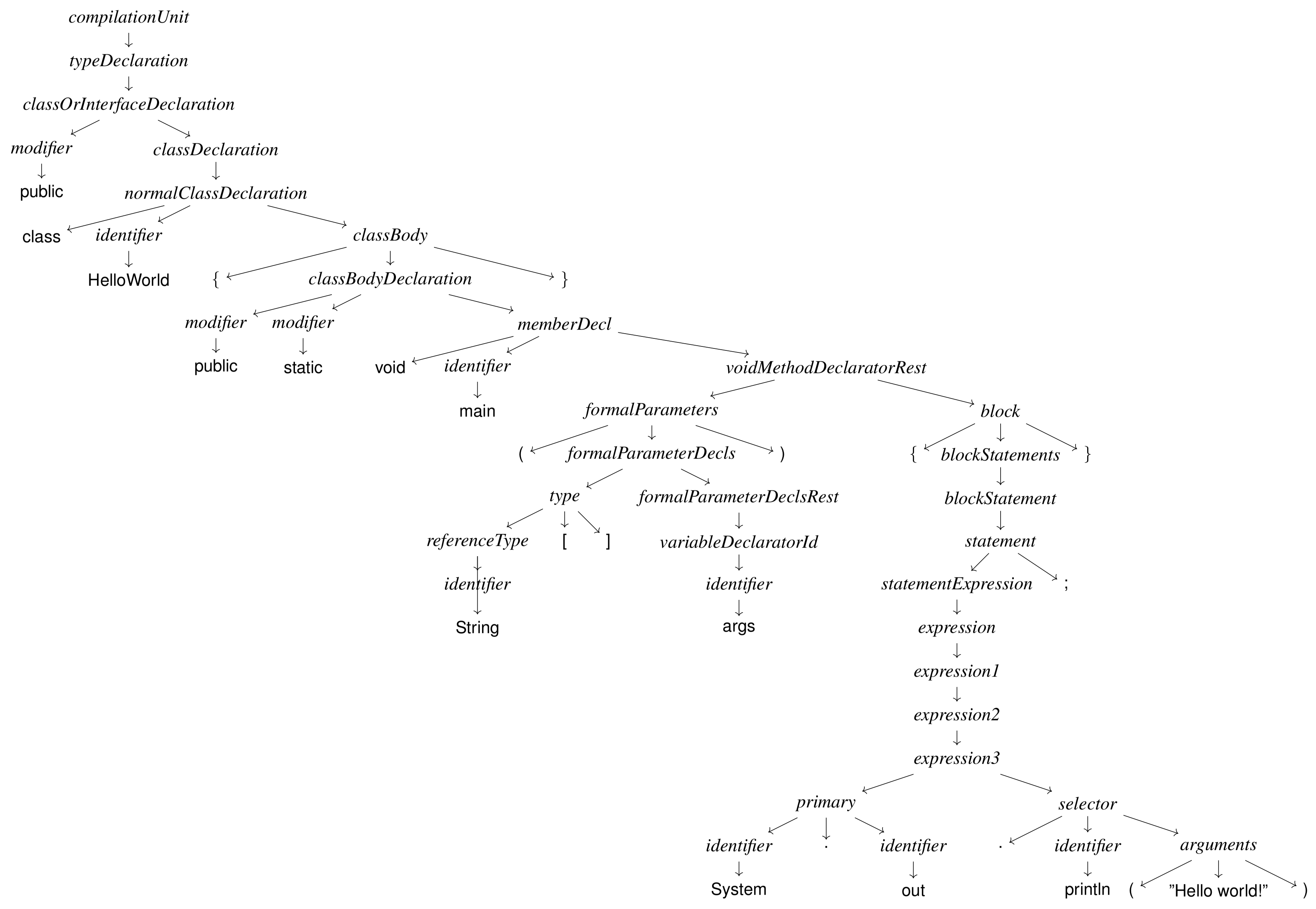


Figure 3.2: The concrete syntax tree for the program of Figure 3.1.

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 LMs 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 *classDeclaration* 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

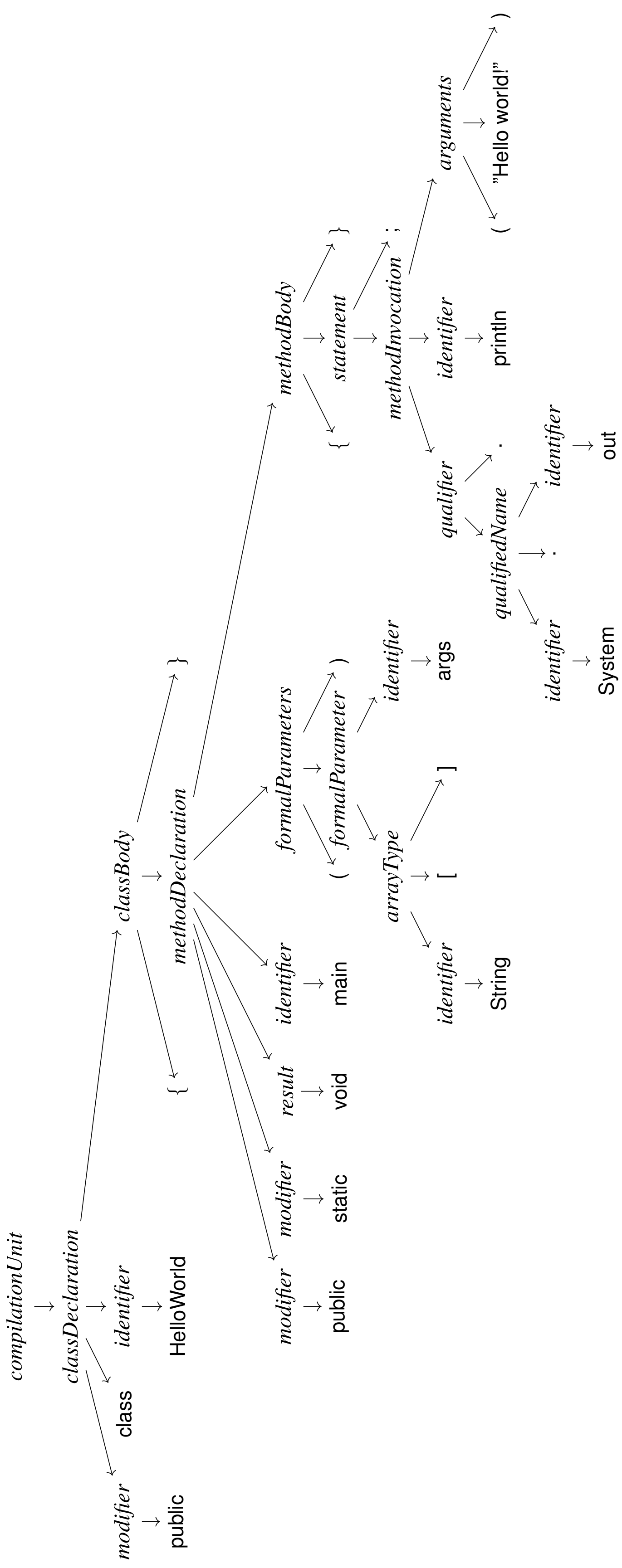


Figure 3.3: The abstract syntax tree derived from the concrete syntax tree of Figure 3.2.

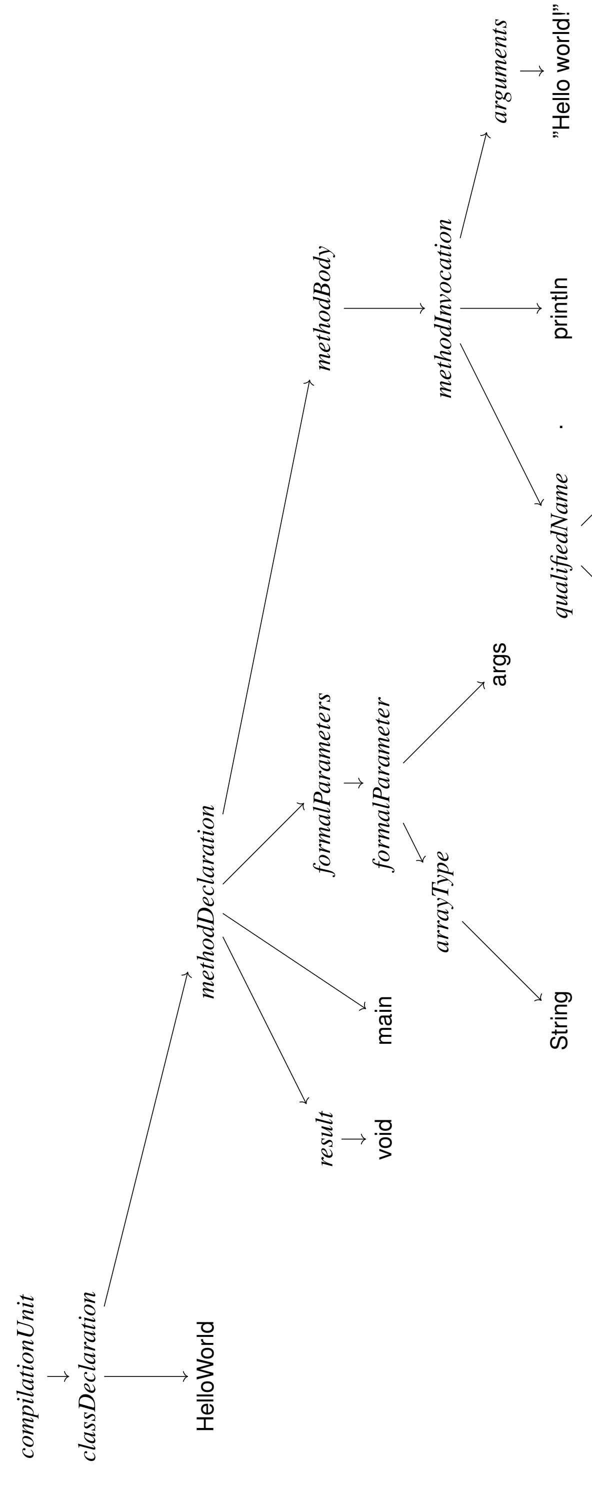


Figure 3.4: A more abstract AST derived from the concrete syntax tree of Figure 3.2.

```
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 logging call. 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 logging call. This will be referred to as Example 2.

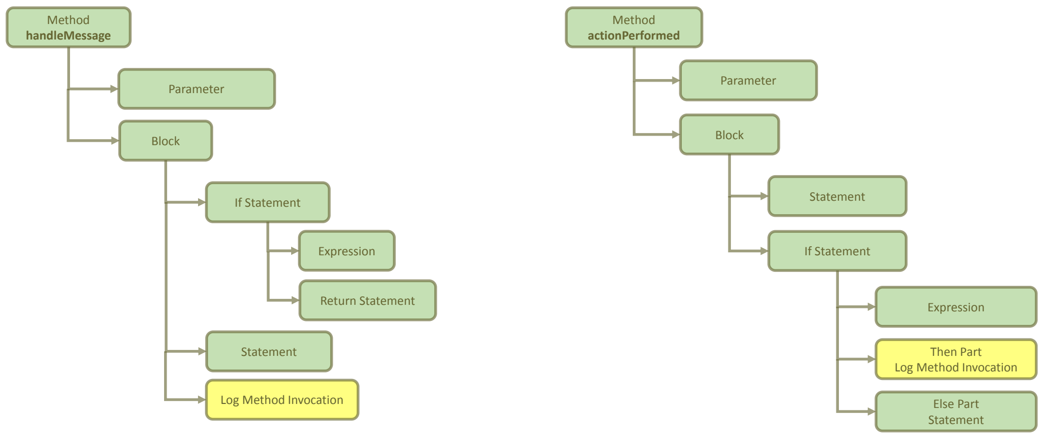


Figure 3.7: Simple AST structure of the examples in Figures 3.5 and 3.6.

child nodes. For example, the *classDeclaration* node in Figure 3.3 can possess multiple *modifier*s.

As an example, the ASTs of the logging calls 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 and anti-instances of a term, as the requirements needed to describe anti-unification theory (and its dual, unification theory).

Definition 3.3.1 (Term)**.** A (first-order) term is defined to be a variable, a constant, or a function symbol followed by a list of terms as the arguments of the function. [Note that function symbols without the subsequent list of terms do not constitute first-order terms.]

Function symbols taking n arguments are called n-ary function symbols; 0-ary function symbols are called constants. The identifiers starting with a lowercase letter are used 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 term:

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

Note that for any term there is a unique, equivalent tree and vice versa: constants and (first-order) variables 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
- $X \leftarrow f \rightarrow c$
- $$\begin{tikzcd} & & & f \arrow[dll] \arrow[d] \arrow[drr] & & & \\ & g \arrow[dl] \arrow[dr] & & Y & & g \arrow[dl] \arrow[dr] & \\ X & & b & & a & & Z \end{tikzcd}$$

Definition 3.3.2 (Substitution)**.** A substitution is a set of mappings, each from a variable to a term.

Definition 3.3.3 (Applying a substitution)**.** Applying a substitution to a term results in the replacement of all occurrences of each variable in the term, by its corresponding term as defined in the substitution.

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

Definition 3.3.4 (Instance & anti-instance)**.** a is an instance of a term X and X is an anti-instance of a, if there is a substitution Θ such that applying Θ to X results in a (i.e., $X \xrightarrow{\Theta} a$).

Definition 3.3.5 (Unifier)**.** A unifier is a common instance of two given terms.

Unification usually aims to create the *most general unifier* (MGU); that is, U is the MGU of two terms such that for all unifiers U' there exists a substitution Θ such that $U \xrightarrow{\Theta} U'$. Unification aims to make a more concrete structure in essence, whereas what we need is a more generalized structure, which leads to the use of the dual of unification, called *anti-unification*.

Definition 3.3.6 (Anti-unifier)**.** X is an anti-unifier (or generalization) for a and b, if X is an anti-instance for a and an anti-instance for b under substitutions Θ_1 and Θ_2, respectively (i.e., $X \xrightarrow{\Theta_1} a$ and $X \xrightarrow{\Theta_2} b$).

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

As an example, the anti-unifier of two given terms $f(X,b)$ and $f(a,Y)$ is the new term $f(X,Y)$, containing common pieces of the two original terms. The variable Y in the anti-unifier $f(X,Y)$

$$\begin{tikzcd}
 & \begin{array}{c}\boxed{f(X,Y)}\\ \text{anti-unifier}\end{array} \arrow[dl, "\Theta_1 = Y \rightarrow b"'] \arrow[dr, "\Theta_2 = X \rightarrow a"] & \\
\boxed{f(X,b)} \arrow[dr, "\Theta'_1 = X \rightarrow a"'] & & \boxed{g(a,Y)} \arrow[dl, "\Theta'_2 = Y \rightarrow b"] \\
 & \begin{array}{c}\boxed{f(a,b)}\\ \text{unifier}\end{array} &
\end{tikzcd}$$

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

$$\begin{tikzcd}
 & \boxed{X} \arrow[dl, "\Theta_1 = X \rightarrow f(a,b)"'] \arrow[dr, "\Theta_2 = X \rightarrow g(a,b)"] & \\
\boxed{f(a,b)} & & \boxed{g(a,b)}
\end{tikzcd}$$

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

can be substituted by the term b to re-create $f(X,b)$ (with $\Theta_1 = Y \rightarrow b$) and the variable X in the anti-unifier can be substituted by the term a to re-create $f(a,Y)$ (with $\Theta_2 = X \xrightarrow{\Theta} a$), as depicted in Figure 3.8. In addition, the unifier $f(a,b)$ of the two terms can be instantiated by applying the substitutions $\Theta'_1 = X \xrightarrow{\Theta} a$ and $\Theta'_2 = Y \xrightarrow{\Theta} b$ on the terms $f(X,b)$ and $f(a,Y)$, respectively.

The MSA should preserve as much of common pieces of both original terms as possible; however, first-order anti-unification fails to capture complex commonalities as it restricts substitutions to only replace first-order variables by terms. That is, when two terms differ in function symbols, first-order anti-unification fails to capture common details of them. 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 would allow us to create the MSA by extending the set of possible substitutions such that variables can be replaced not only by terms but also by function symbols in

$$\begin{tikzcd}
& \boxed{X(a,b)} \arrow[dl, "\Theta_1 = X \to f"'] \arrow[dr, "\Theta_2 = X \to g"] & \\
\boxed{f(a,b)} & & \boxed{g(a,b)}
\end{tikzcd}$$

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

order to retain the detailed commonalities. For example, the higher-order anti-unifier of the terms $f(a,b)$ and $g(a,b)$ is $X(a,b)$ as depicted in Figure 3.10.

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 different 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 extended structures to determine structural equivalences. For example, we have introduced an equivalence equation $=_E$, such that $f(X,Y) =_E f(Y,X)$ to indicate that the ordering of arguments does not matter in our context.

We have also introduced a theory, called NIL-theory, that adds the concept of a NIL structure, which permits a structure to be equated with nothing, and defines an equivalence equation $=_E$ for it. The NIL structure can be used to anti-unify two structures when a substructure exists in one but is missing from the other. However, some requirements should be taken to avoid the overuse of NIL structures such that the original structures must have common substructures but vary in the size for dissimilar substructures. For example, we can anti-unify the two structures b and $f(a,b)$

$$\begin{tikzcd} & \boxed{X(Y,b)} \arrow[dl, "\Theta_1 = (X \to f, Y \to a)"'] \arrow[dr, "\Theta_2 = (X \to \mathsf{NIL}, Y \to \mathsf{NIL})"] & \\ \boxed{f(a,b)} & & \boxed{\mathsf{NIL}(\mathsf{NIL}, b) =_E b} \end{tikzcd}$$

Figure 3.11: Higher-order anti-unification modulo theories of the terms $f(a, b)$ and b.

through the application of NIL-theory by creating the term $\mathsf{NIL}(\mathsf{NIL}, b)$ which is $=_E$ to $f(b)$ and anti-unifying $\mathsf{NIL}(\mathsf{NIL}, b)$ with $f(a, b)$ as depicted in Figure 3.11.

We have also defined a set of equivalence equations to incorporate semantic knowledge of structural equivalences supported by the Java language specification, as it provides various ways to define the same language specifications. These theories should be applied on higher-order extended structures to anti-unify AST structures that are not identical but are semantically equivalent. For example, consider **for**- and **while**-statements that are two types of looping structure in Java programming language: they have different syntax but semantically cover the same concept. Let us look at the code snippets **for**(i=0;i<10;i++) and **while**(i<10), whose ASTs can be represented as **for**(*initializer*(i, 0); *lessThanExpression*(i, 10); *updaters*(*postIncrementExpression*(i))) and **while** (*lessThanExpression*(i, 10)), respectively. We could define an equivalence equation $=_E$ that allows the anti-unification of **for**- and **while**-statements. We also need to utilize the NIL-theory to handle the varying number of arguments as the **for**-loop has three arguments 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 $=_E$ 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 3.12.

However, defining complex substitutions in higher-order anti-unification modulo theories results in losing the uniqueness of the MSA. For example, consider the terms $f(g(a, e))$ and $f(g(a, b), g(d, e))$. As described in Figure 3.13, two MSAs exist for these terms: we can anti-unify $g(a, e)$ and $g(a, b)$ to create the anti-unifier $g(a, X_0)$ and anti-unify $g(d, e)$ with the NIL structure to create the anti-unifier $Y(Z, X_1)$; or we can anti-unify $g(a, e)$ and $g(d, e)$ to create the anti-unifier $g(X_0, e)$

$$\begin{tikzcd}[column sep=small]
& \boxed{\begin{array}{l} V_0(V_1(V_2, V_3),\\ \quad \textit{lessThanExpression}(\mathsf{i},\ \mathsf{10}),\\ \quad V_4(V_5(V_2)))\end{array}} \arrow[dl, "\Theta_1"'] \arrow[dr, "\Theta_2"] & \\
\boxed{\begin{array}{l} \mathsf{for}(\textit{initializer}(\mathsf{i},\ \mathsf{0}),\\ \quad \textit{lessThanExpression}(\mathsf{i},\ \mathsf{10}),\\ \quad \textit{updater}(\textit{postIncrementExpression}(\mathsf{i})))\end{array}} & & \boxed{\begin{array}{l} \mathsf{while}(\mathsf{NIL}(\mathsf{NIL},\ \mathsf{NIL}),\\ \quad \textit{lessThanExpression}(\mathsf{i},\ \mathsf{10}),\\ \quad \mathsf{NIL}(\mathsf{NIL}(\mathsf{NIL})))\end{array}}
\end{tikzcd}$$

Figure 3.12: 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: $\Theta_1 = (V_0 \rightarrow$ for$, V_1 \rightarrow$ *initializer*$, V_2 \rightarrow$ i$, V_3 \rightarrow 0, V_4 \rightarrow$ *updater*$, V_5 \rightarrow$ *postIncrementExpression*$)$; and $\Theta_2 = (V_0 \rightarrow$ while$, V_1 \rightarrow$ NIL$, V_2 \rightarrow$ NIL$, V_3 \rightarrow$ NIL$, V_4 \rightarrow$ NIL$, V_5 \rightarrow$ NIL$)$

$$\begin{tikzcd}[column sep=small]
& \boxed{f(g(X_0, e), Y(Z, X_1))} \arrow[dl, "\Theta_1 = (X_0 \rightarrow d, Y \rightarrow g, Z \rightarrow a, X_1 \rightarrow b)"'] \arrow[dr, "\Theta_2 = (X_0 \rightarrow a, Y \rightarrow \mathsf{NIL}, Z \rightarrow \mathsf{NIL}, X_1 \rightarrow \mathsf{NIL})"] & \\
\boxed{f(g(a, b), g(d, e))} & & \boxed{f(g(a, e))}
\end{tikzcd}$$

$$\begin{tikzcd}[column sep=small]
& \boxed{f(g(a, X_0), Y(Z, X_1))} \arrow[dl, "\Theta_1 = (X_0 \rightarrow b, Y \rightarrow g, Z \rightarrow d, X_1 \rightarrow e)"'] \arrow[dr, "\Theta_2 = (X_0 \rightarrow e, Y \rightarrow \mathsf{NIL}, Z \rightarrow \mathsf{NIL}, X_1 \rightarrow \mathsf{NIL})"] & \\
\boxed{f(g(a, b), g(d, e))} & & \boxed{f(g(a, e))}
\end{tikzcd}$$

Figure 3.13: Ambiguous higher-order anti-unification modulo theories of the terms $f(g(a, b), g(d, e))$ and $f(g(a, e))$, creating multiple MSAs.

and anti-unify $g(a, b)$ with the NIL structure to create the anti-unifier $Y(Z, X_1)$.

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

3.5 The Jigsaw 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 one fragment can be integrated to the other one for small-scale code reuse.

In general, the proposed approach by 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 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 references to variables in the original system by the references to variables in the target system. 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 logging calls 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.

3.6 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 insertion and deletion by declaring equivalence with the NIL structure, as well as other 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.

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 3.5). Then, I extended the CAST structure to *anti-unifier ASTs* (AUASTs) structure that would allow the creation of 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 discuss 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.

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

They compute the number of matched elements between the two ASTs A and B via the DETERMINE-MATCHES algorithm. As mentioned in Section 3.5, 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 3.5 (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 similarity 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 (line 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 ← 0
 2: if LOOKUP-COMPARATOR(type[root[A]], type[root[B]]) then
 3:     if A instanceof Leaf-Node then
 4:         matches ← MATCHES(A, B)
 5:     else if A instanceof Non-Leaf-Node then
 6:         for childA ∈ children[A] do
 7:             max ← 0
 8:             for childB ∈ children[B] do
 9:                 matches ← DETERMINE-MATCHES(childA, childB)
10:                 if matches > max then
11:                     max ← matches
12:                 end if
13:             end for
14:         end for
15:         matches ← 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 3.5, the proposed approach by 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 DETERMINE-MATCHES 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: $sim \leftarrow matches / \max\{|A|, |B|\}$
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 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

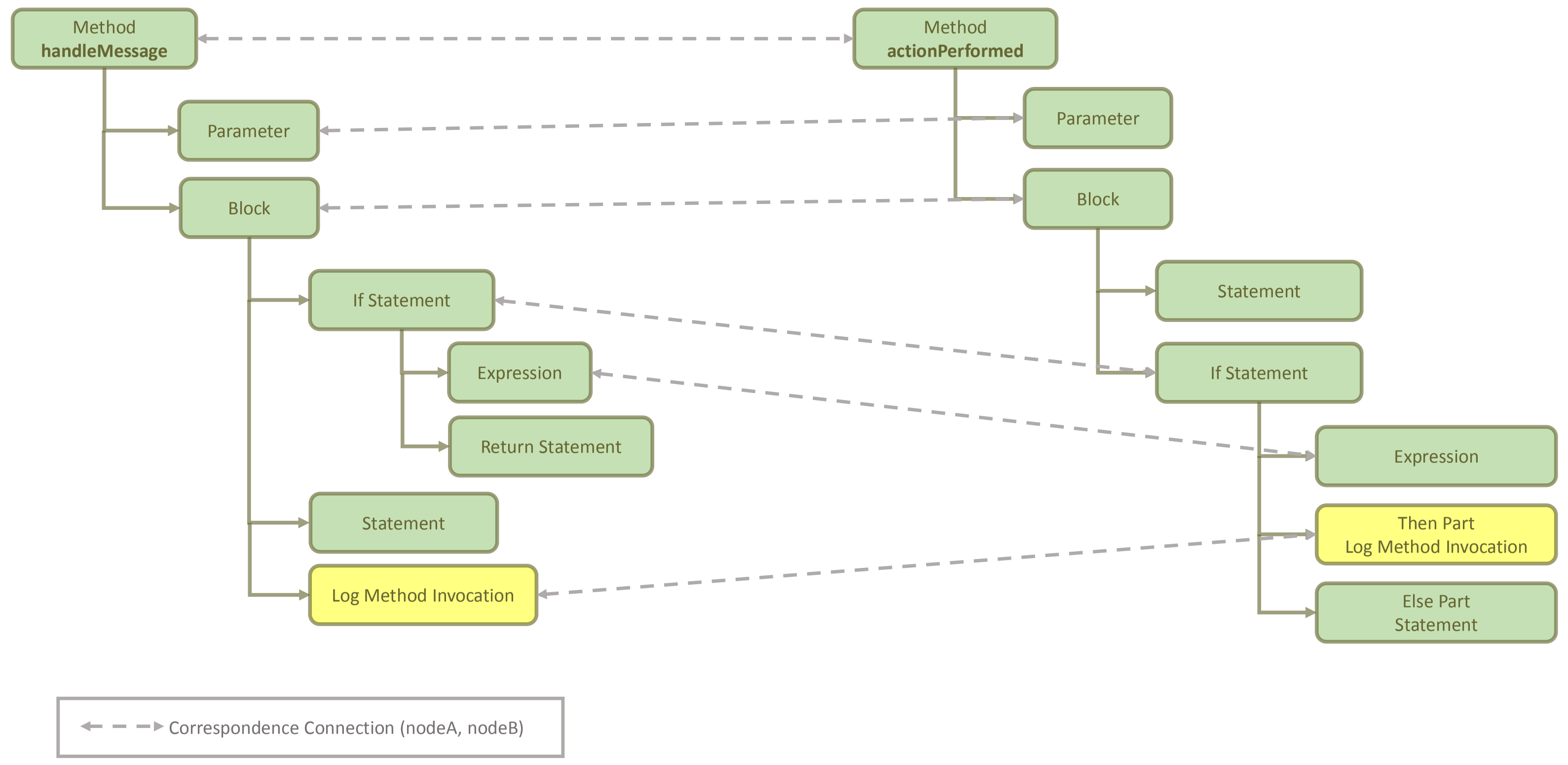


Figure 4.1: Simple CAST structure of the examples in Figures 3.5 and 3.6. The links between the CAST nodes indicate the structural correspondence connections.

the AUASTs of logging calls 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.

$$\begin{aligned}\Theta_1 = (X \rightarrow \text{ WARNING}, Y \rightarrow \ & \textit{additionExpression}(\\ & \textit{methodCall}(\textit{simpleName}(\text{getClassName}), \textit{arguments}()), \\ & \textit{stringLiteral}(\text{"should extend ..."}), \\ & \textit{methodCall}(\textit{simpleName}(\text{handleMessage}), \textit{arguments}()))) \end{aligned} \tag{4.2}$$

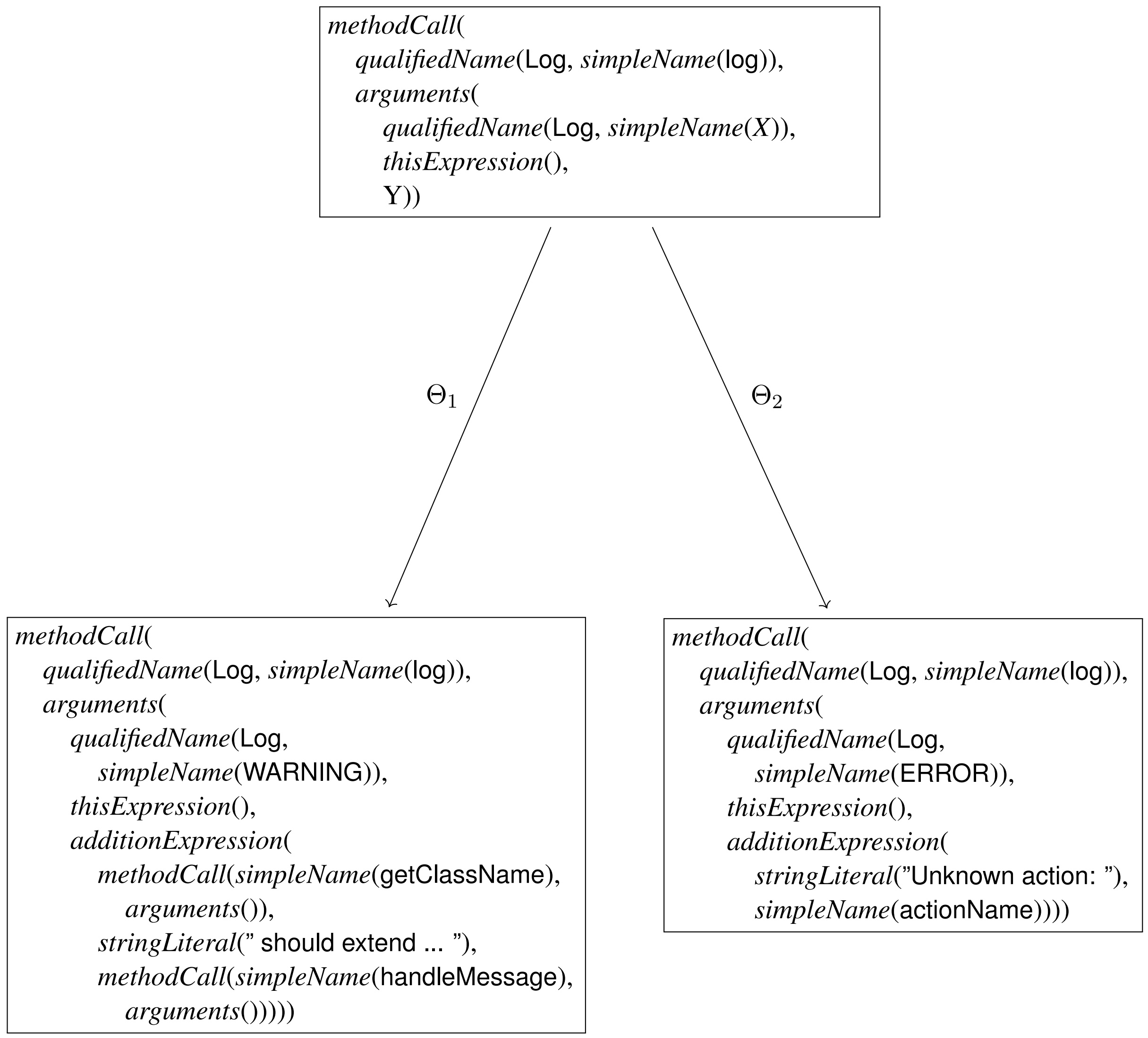
$$\begin{aligned}\Theta_2 = (X \rightarrow \text{ ERROR}, Y \rightarrow \ & \textit{additionExpression}(\\ & \qquad \textit{stringLiteral}(\text{"Unknown action: "}), \\ & \qquad\qquad \textit{simpleName}(\text{actionName}))) \end{aligned} \tag{4.3}$$

4.4 An assessment of the Jigsaw framework

To validate the effectiveness of the proposed approach by Cottrell et al. [2008], I applied the Jigsaw framework to compare the ASTs of a sample set of logged methods 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 **[RW: Which version?]**), 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 logging calls. My tool extracts all LMs within the source code of this program using the Eclipse JDT framework. I



```
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 logging calls 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.

selected a subset consisting of 9 LMs, showing varying levels of similarity on manual examination, as a test suite throughout this study (see Table 7.1). The EditBus.send(..) method contains two logging calls. To handle this case I split it into two cases: Case 3 contained the first logging call while the second one was removed; Case 4 contained only the second logging call.

I describe my approach for LMs containing multiple logging calls 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 logging call. 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 logging call 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()){ ...}
		[RW: ???]

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 logging call but is syntactically relevant.

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.

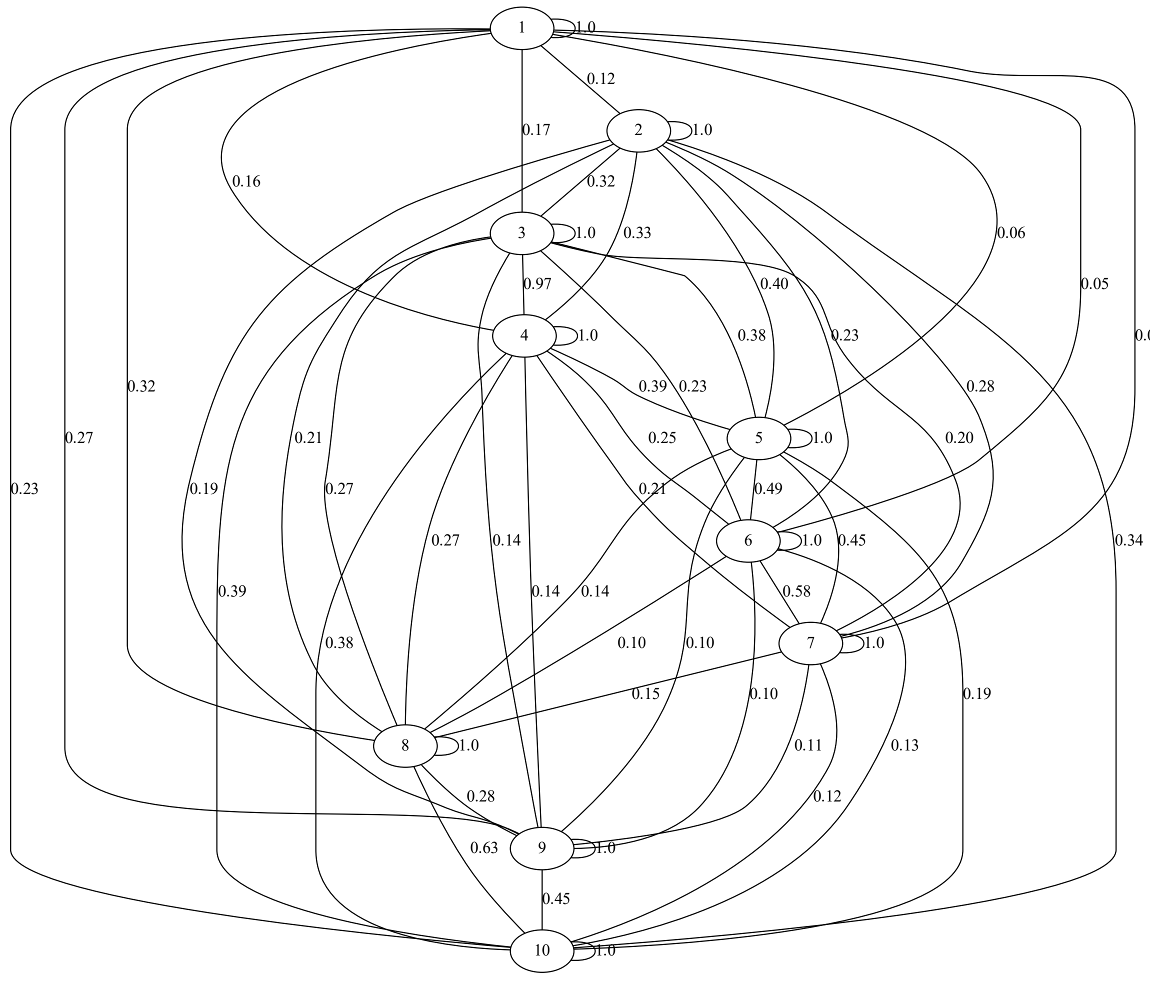


Figure 4.5: A similarity graph representing pairwise Jigsaw similarities between LMs shown in Table 7.1.

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 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

In Chapter 3, I provided background information on higher-order anti-unification modulo theories, a theoretical framework that can be used to construct a generalization from two given ASTs. In Chapter 4, I presented and tested the Jigsaw framework proposed by Cottrell et al. [2008] to construct the CAST structure, where each node holds a list of candidate structural correspondences. I next extended the CAST structure to an AUAST that would allow the creation of an anti-unifier. I now consider how these frameworks 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 should 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 should be taken, which is the anti-unification of each AUAST node with its best correspondence (Section 7.2). Furthermore, I developed an algorithm to measure similarity between the usage of logging calls in methods based on the selected correspondences (Section 5.2). An overview of the proposed anti-unification approach is shown in Figure 5.1.

To evaluate my approach, I developed the anti-unifier-building tool atop Jigsaw, and conducted

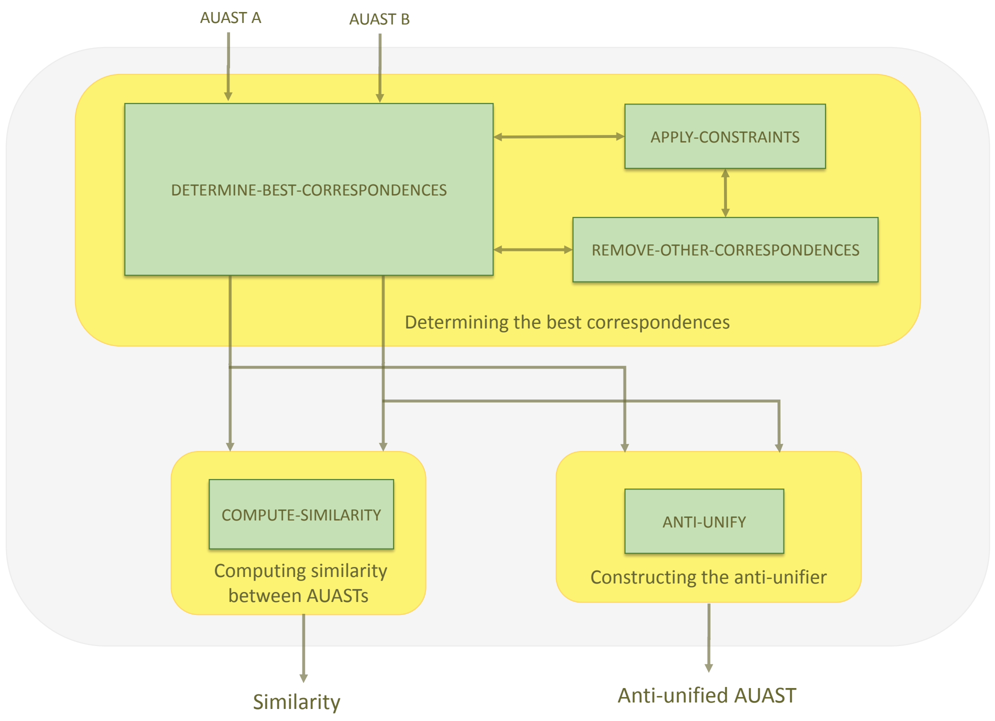

AUAST A
AUAST B
DETERMINE-BEST-CORRESPONDENCES
APPLY-CONSTRAINTS
REMOVE-OTHER-CORRESPONDENCES
Determining the best correspondences
COMPUTE-SIMILARITY
Computing similarity between AUASTs
ANTI-UNIFY
Constructing the anti-unifier
Similarity
Anti-unified AUAST

Figure 5.1: Overview of the anti-unification process.

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.

5.1 Determining the best correspondences

As discussed in Section 3.4, the complexity of determining the best anti-unifier is undecidable in general. 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. Therefore, 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, some constraints should be applied 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 $nodeA \in A$ do
 4:     for $corr \in corrs[nodeA]$ 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 logging calls, 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 logging call should either be anti-unified with another logging call or should be anti-unified with "nothing".

This constraint creates a further constraint:

Constraint 2. A structure containing a logging call should be anti-unified with a corresponding structure containing another logging call 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 a property to the AUAST node 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, using the REMOVE-OTHER-CORRESPONDENCES algorithm (lines 1–9).

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

```
APPLY-CONSTRAINTS($list$)
for $corr \in list$ do
    $nodeA \leftarrow nodeA[corr]$
    $nodeB \leftarrow nodeB[corr]$
    if ($nodeA$ instanceof Logged) & ($nodeB$ instanceof Non-Logged) then
        REMOVE-OTHER-CORRESPONDENCES($corr, list$)
    else if ($nodeA$ instanceof Non-Logged) & ($nodeB$ instanceof Logged) then
        REMOVE-OTHER-CORRESPONDENCES($corr, list$)
    end if
end for
```

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); 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.

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 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 logging calls. 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.

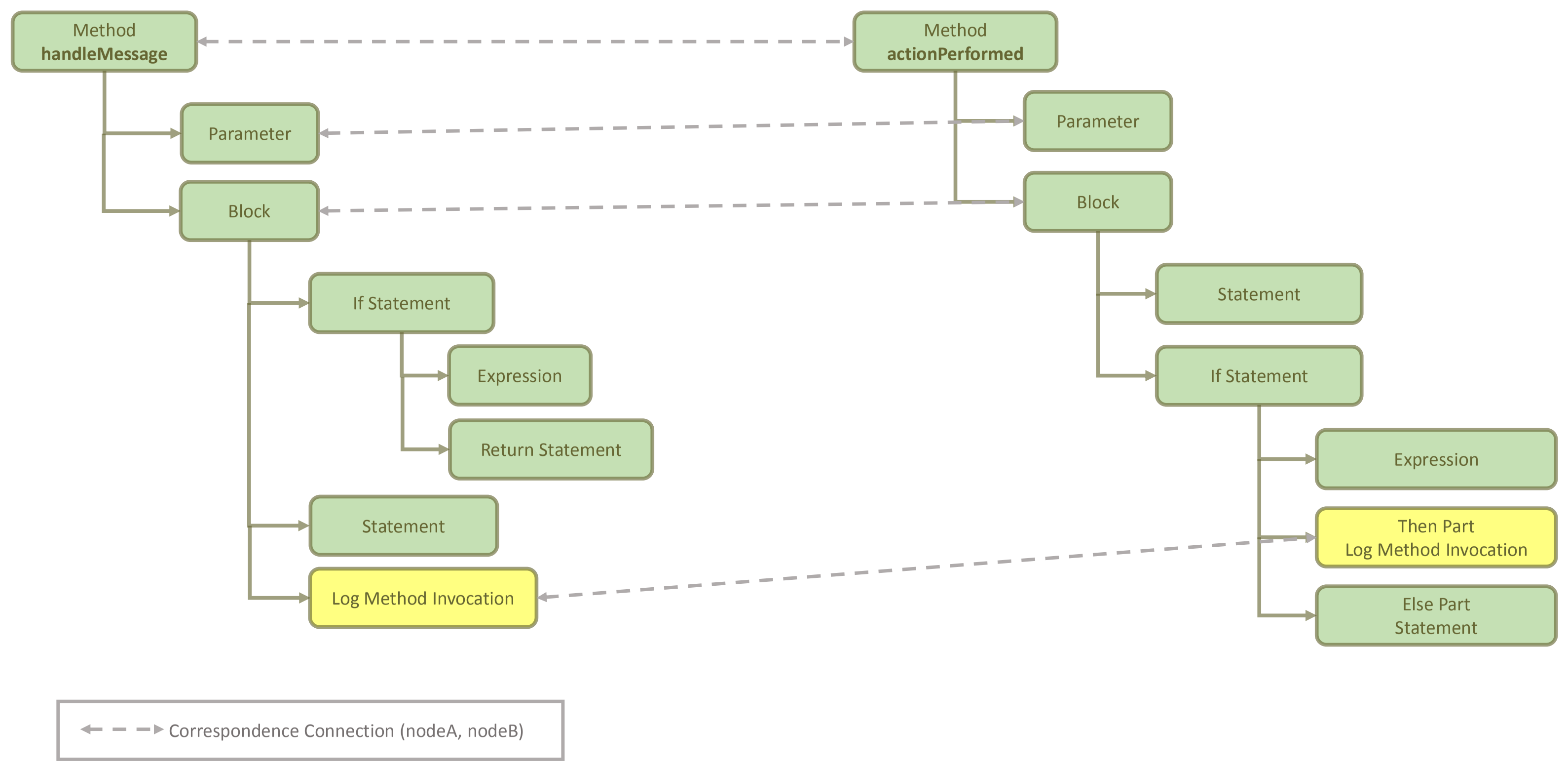


Figure 5.2: Simple AUAST structure of examples in Figures 3.5 and 3.6. The links between AUAST nodes indicate structural correspondences selected as the best fit using the DETERMINE-BEST-CORRESPONDENCES algorithm.

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
```

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 of leaf nodes, the number of matches is computed using the heuristics proposed by Cottrell et al. [2008] (previously described in Section 3.5) (lines 2–3). 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).

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

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

node with its best correspondence. The new anti-unified structure is a generalization of two original structures, called anti-unifier, where common structural properties are represented by copies and differences are represented by structural variables. The variables may be inserted in place of any node in AUAST, including both subtrees and leaves, and can be substituted with proper original substructures to regain original structures.

Anti-unification of two AUASTs is performed via the ANTIUNIFY algorithm. If the two AUASTs are of 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 children 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.

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 $root[A]$ instanceof Leaf-Node then
 3:     $antiunifier \leftarrow$ CONSTRUCT-ANTIUNIFIER($root[A], root[B]$)
 4: else if $root[A]$ 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($childA, node[bestCorr[childA]]$)
10:         end if
11:         APPEND($child, children[antiunifier]$)
12:     end for
13: end if
14: if $root[B]$ 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 $antiunifier$
```

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 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 logged methods, I developed an algorithm to construct a generalization view of the anti-unifier. Figure 5.3 shows a simple detailed view I used to represent the anti-unifier constructed from the AUASTs of logged methods of Examples 1 and 2, where "a-or-b" represents a structural variable that must be substituted with either the a or b substructures to recover each original structure, and "a" represents that the a substructure is common between the two AUASTs.

To create the detailed view of the structural generalization from two given AUASTs, I applied the ANTIUNIFY algorithm presented in Section 7.2 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 refJDT). 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 and added to the anti-unifier's structural properties.

```
method(modifiers(public),returntype2(void),name(actionPerformed-or-
handleMessage),parameters(type(ActionEvent-or-EBMessage),name(evt-or-
message)),body(statements(if-or-NIL(expression-or-NIL(leftoperand-or-NIL(action-or-NIL),==-
or-NIL),thenstatement-or-NIL(statements-or-
NIL(expression(expression(Log),name(log),arguments(leftoperand(actionName-or-
message),+,rightoperand("is an unknown action"-or-" is
empty"),qualifier(Log),name(WARNING))))),elsestatement-or-NIL(expression-or-
NIL(expression-or-NIL(context-or-NIL),name-or-NIL(invokeAction-or-NIL),arguments-or-
NIL(evt-or-NIL,action-or-NIL)))),type-or-NIL(EditAction-or-NIL),fragments-or-NIL(name-or-
NIL(action-or-NIL),initializer-or-NIL(expression-or-NIL(context-or-NIL),name-or-NIL(getAction-
or-NIL),arguments-or-NIL(actionName-or-NIL))),expression-or-NIL(lefthandside-or-
NIL(seenWarning-or-NIL),=-or-NIL,righthandside-or-NIL(true-or-NIL)),if-or-NIL(expression-or-
NIL(seenWarning-or-NIL))))),method(modifiers(public-or-NIL,protected-or-NIL),name(Wrapper-
or-EBPlugin),parameters-or-NIL(type-or-NIL(ActionContext-or-NIL),name-or-NIL(context-or-
NIL),type-or-NIL(String-or-NIL),name-or-NIL(actionName-or-NIL)),body(statements-or-
NIL(expression-or-NIL(lefthandside-or-NIL(name-or-NIL(context-or-NIL)),=-or-
NIL,righthandside-or-NIL(context-or-NIL)),expression-or-NIL(lefthandside-or-NIL(name-or-
NIL(actionName-or-NIL)),=-or-NIL,righthandside-or-NIL(actionName-or-NIL)))))
```

Figure 5.3: Simple detailed view of the anti-unifier constructed from the AUASTs of Examples 1 and 2.

5.4.2 Java methods containing multiple logging calls

There might be some cases in which our approach is not able to anti-unify logging calls 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, log1 should be anti-unified with log3, and log4 should be anti-unified with NIL since there is no corresponding logging call in the body of **if** statement 1, while there is a corresponding logging call for log4 in the body of **if** statement 2 (log2).

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

Figure 5.4: A Java method that utilizes multiple logging calls.

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

Figure 5.5: A Java method that utilizes multiple logging calls.

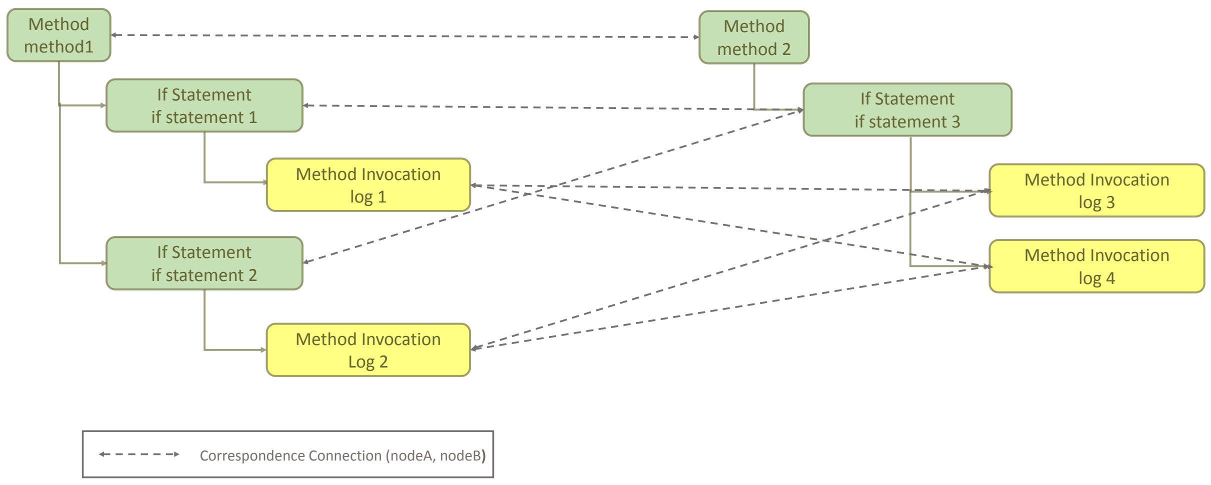


Figure 5.6: Simple AUAST structure of examples in Figures 5.4 and 5.5. Links between AUAST nodes indicate candidate structural correspondences detected by the Jigsaw framework.

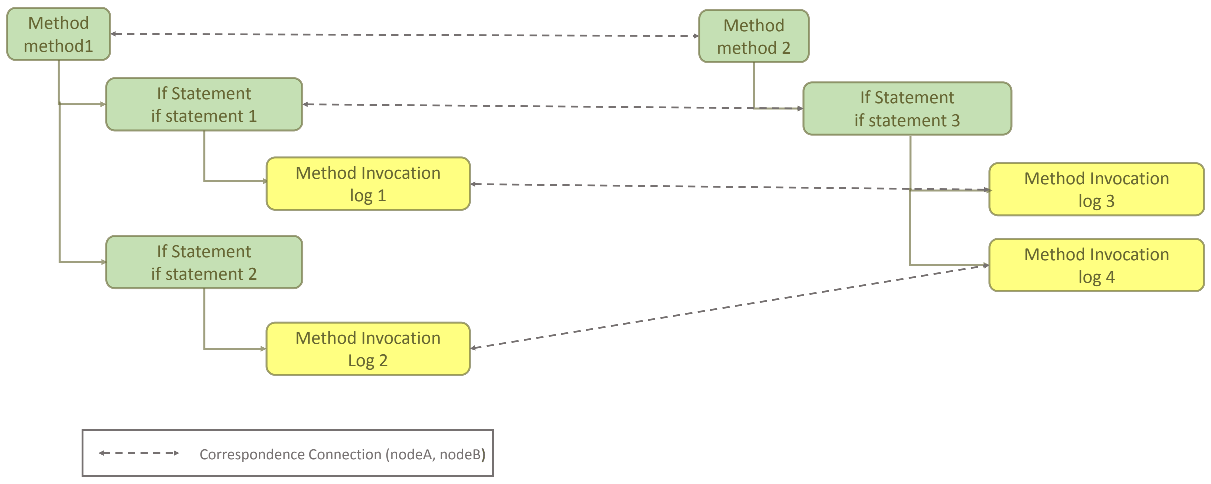


Figure 5.7: Simple AUAST structure of examples in Figures 5.4 and 5.5. Links between AUAST nodes indicate structural correspondences selected as the best match using our greedy algorithm.

To handle these cases, we can split them into more than one case, where each LM contains only one logging call. To do so, we need to create a copy of the LM for each logging call 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: Create multiple copies of the LM in Figure 5.4 for each logging call.

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

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

Figure 5.9: Create multiple copies of the LM in Figure 5.5 for each logging call.

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, we manually attempted to create the detailed anti-unifier view for each pair of LMs in the test suite (55 test cases in total). We first identified corresponding and non-corresponding Java elements for each LM pair with a focus on preventing the correspondence of logging calls with anything else and then represented the anti-unifier in the detailed view (i.e., formatted as in Figure 5.3). We 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. We 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 logging calls and to measure the similarity between the two LMs. Furthermore, we used EclEmma, which is a Java code coverage tool for Eclipse, to measure the

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 coverage. Test coverage is defined as a measure of the completeness of the set of test cases.

5.5.2 Results

We present the results of our 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 our tool that were found to be corresponded by our manual examination as well, and the number of LOC detected as corresponded by our tool but were not found to be corresponded in our manual inspection. We also present the percentage of the correct corresponding LOC to the total number

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 logging calls for the 55 test cases.

of LOC of the two LMs. "Similarity" refers to the similarity that is computed based on the the detected correspondences. It is calculated using both our tool and manual experiment.

In Case 8, the rootEntry(..) method contains a nested **if**-statement enclosing a logging call and the actionPerformed(..) method contains an **if**-statement enclosing another logging call. 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 our 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 as well. 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. Our anti-unifier-building tool succeeded in detecting correspondences with special attention to anti-unifying logging calls 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.

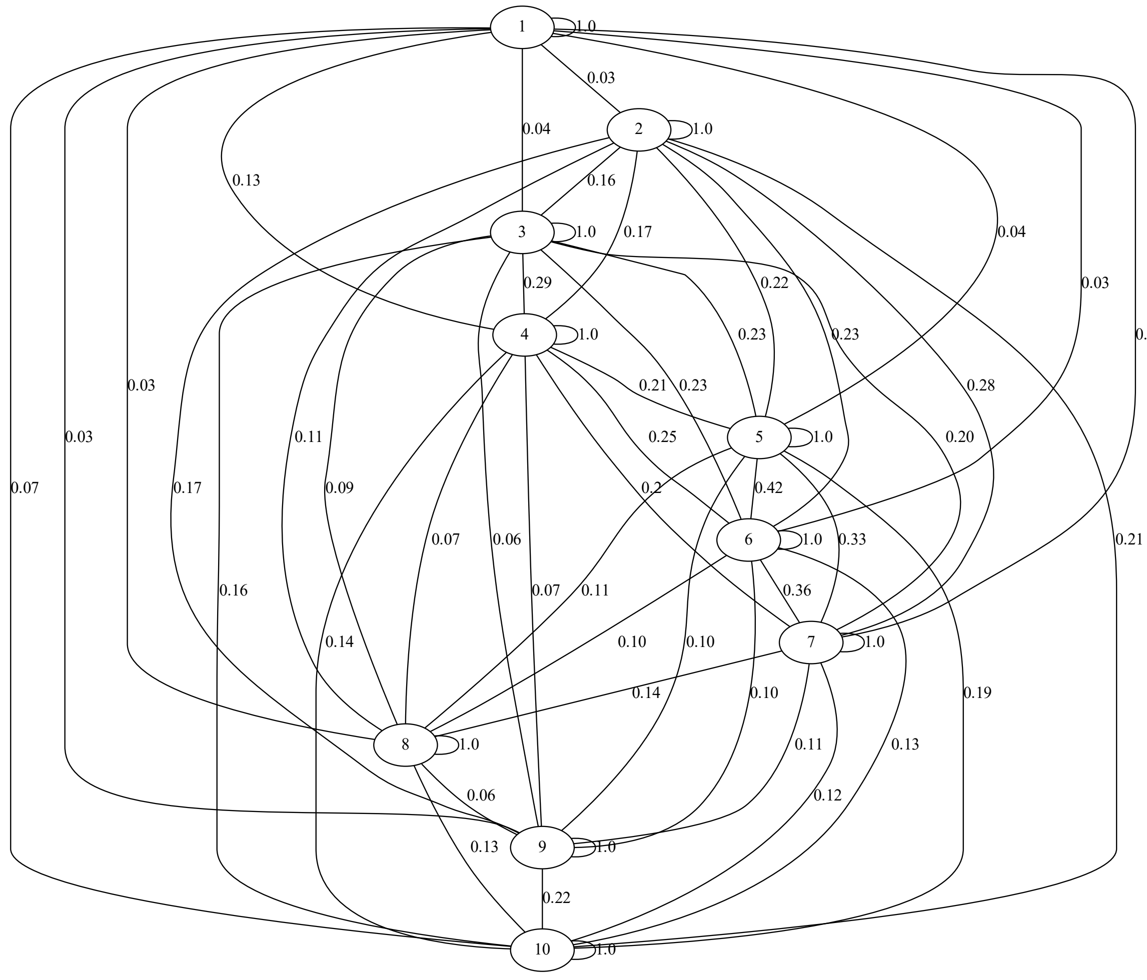


Figure 5.12: A similarity graph representing pairwise similarities calculated by our tool between LMs shown in Table 7.1.

5.6 Summary

I have presented an approach for constructing a generalization from AUASTs of two LMs with special attention to logging calls 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 our anti-unification algorithm and the tool support in constructing an anti-unifier from AUASTs of each LM pair in our test suite with a focus on logging calls and measuring similarity between them.

Chapter 6

Clustering

In Chapter 5, I described my anti-unification algorithm to construct an anti-unifier from the AUASTs of a pair of LMs, paying special attention to logging calls. Recall that the general point of this study is to provide a concise description of where logging calls happen in source code by constructing structural generalizations that represent the detailed commonalities and differences between AUASTs of LMs. To this end, we should develop an algorithm that:

- clusters Java methods showing different usages of logging calls into separate clusters; and
- abstracts AUASTs of LMs of each group into a structural generalization representing the commonalities and differences between them.

Clustering is the classification of a collection of unlabelled data items into meaningful groups [**?**], where category labels are obtained from the similarities between data items. Therefore, before the use a clustering algorithm to classify a set of AUASTs, I need to first define a similarity metric. To develop a measure of similarity between AUASTs, I used the similarity function described in Section 5.2. To perform clustering on a set of AUASTs of LMs, I developed a modified version of an agglomerative hierarchical clustering algorithm suited to my application. The clustering algorithm is a bottom-up approach that starts with singleton clusters, each contains 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 (Section **??**) and conducted an experimental study through the application of it on the test suite introduced in Section 4.4.1. I will describe my experimental study and discuss the results in Section **??**.

6.1 The hierarchical clustering algorithm

To anti-unify a set of AUASTs, I have developed a modified version of an agglomerative hierarchical clustering algorithm as described below:

1. Start with singleton clusters, each contains one AUAST
2. Create a similarity matrix by computing pairwise similarities between clusters
3. Find the closest clusters (a cluster pair with maximum similarity)
4. Merge the closest cluster pair and replace them with a new cluster containing the anti-unifier of AUASTs of the two clusters
5. Update the similarity matrix by computing the similarity between new cluster and all remaining clusters

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

The hierarchical algorithm uses a similarity matrix. It employs a $n \times n$ similarity matrix for a set of n AUASTs, where an element in row i and column j represents the similarity between the i^{th} and the j^{th} clusters. In this algorithm, the similarity between a pair of clusters is defined as the similarity between their AUASTs, which is computed through the algorithm described in Section 5.2. However, to prevent the combination of a cluster pair when the usage of logging is different, I adjusted the similarity between them to zero. That is, if the anti-unification of AUASTs of a cluster pair does not allow the anti-unification of log statements with each other (as the structures enclosing them are not corresponded), they should be in separate clusters. I also used the anti-unification algorithm described in Section 7.2 to construct structural generalizations.

Figure 6.2 illustrates the clustering process for a sample set of 5 AUASTs using the initial similarity matrix depicted in Figure 6.1. In the first and second iterations, clusters 1 and 2, and then clusters 4 and 5 are selected as the closest clusters, merged, and replaced by clusters 6 and 7, respectively. If threshold value is determined as Threshold $A = 0.20$, the process will be terminated as the similarity between the closest clusters, which are clusters 3 and 6 is below this threshold; otherwise, these clusters will be merged and replaced by cluster 8. However, the similarity between AUASTs of clusters 7 and 8 is zero, and thus they should not be merged with each other. In this study, the similarity threshold is determined through informal experimentation.

$$similarity = \begin{bmatrix} 1.00 & & & & \\ 0.28 & 1.00 & & & \\ 0.12 & 0.17 & 1.0 & & \\ 0.00 & 0.00 & 0.00 & 1.0 & \\ 0.00 & 0.00 & 0.00 & 0.21 & 1.00 \end{bmatrix}$$

Figure 6.1: The similarity matrix for a sample set of 5 AUASTs.

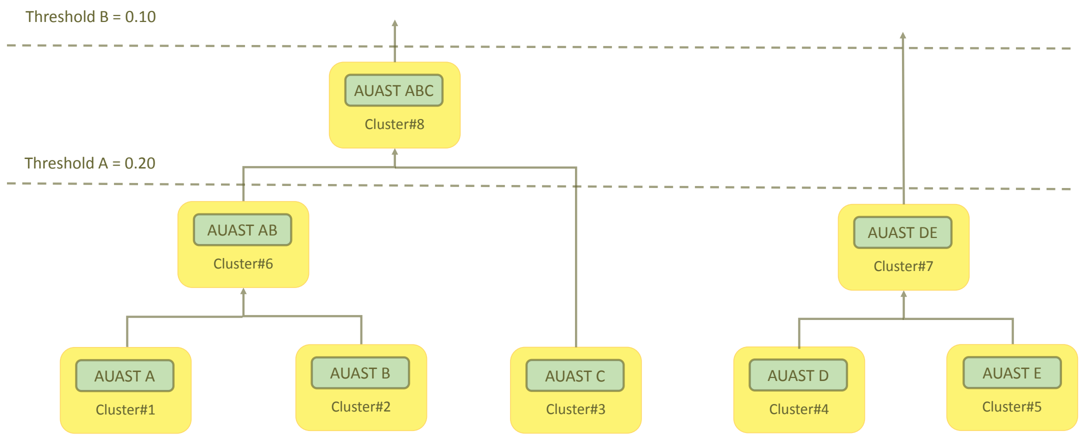


Figure 6.2: The agglomerative hierarchical clustering process on a sample set of 5 AUASTs using the initial similarity matrix shown in Figure 6.1. The threshold value indicates the number of clusters we will come up with.

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 7.1. 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 a structural generalization for each cluster.

6.2.1 Setup

I manually attempted to perform the hierarchical clustering on the set AUASTs of LMs in the test suite and constructed the detailed anti-unifier view for each cluster. Anti-unifiers were discarded when the anti-unification of LMs did not allow the anti-unification of logging calls with one another, as the Java elements enclosing them were not found to be corresponded. I also measured the level of similarity between AUASTs in each cluster by computing the ratio of common Java elements in the detailed anti-unifier view to the total number of Java elements of all AUASTs in that cluster. I also ran the clustering tool on the set of AUASTs to classify them using the similarity measurement.

6.2.2 Results

I present the results of my analysis in Table 6.3. The analysis of the output has been divided into three categories: correspondence, similarity, and separateness. The analysis of correspondence and similarity was described in Section 5.5.2. ”Separateness” refers to my tools’ ability to cluster Java method with different usages of logging calls into separate groups, and the ones with similar usages of logging calls into the same clusters. By similar usage of logging calls, I mean the anti-unification of AUASTs of Java methods in each cluster allows the anti-unification of their log statement nodes with one another, as well.

Cluster	Correspondence		Similarity		Separateness
	Correct (%)	Incorrect	human	tool	
1	28(100)	0	0.09	0.09	✓
3	24(92)	2	0.19	0.2	✓
2	9(100)	0	0.25	0.25	✓

Figure 6.3: Results from applying the clustering tool to the test suite described in Table 7.1

The clustering tool succeeded in detecting the separateness amongst AUASTs of test cases correctly. Clusters 1, 2, and 3 contain logged Java methods of cases (1, 3, 5, 8), (4, 6, 7), and (2, 9, 10), respectively, as detected by my manual inspection. It also successfully calculated the similarity between LMs of 2 clusters out of 3. In Cluster 2, the error in detecting correspondences originated from the previous study and propagated to the clustering study. However, it is trivial (0.01) and would have a low impact on our final results.

6.3 Summary

I have presented a modified version of the agglomerative hierarchical clustering algorithm to classify Java methods with different usages of logging calls into separate groups. This algorithm is implemented as an Eclipse plug-in that takes a set of AUASTs of LMs, clusters them via the application of the anti-unifier building tool to measure pairwise similarities between the AUASTs of cluster pairs, and generates a structural generalization 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 detailed view of structural generalizations 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 five popular open-source software systems: Apache Tomcat, Hibernate ORM, Apache Camel, Apache Solr, and .... 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 a 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 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 level, 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 five software systems used in the characterization study.

My proof-of-concept implementation takes the source code of these systems as input, extracts the ASTs of their LMs, applies the proposed algorithm to construct AUASTs, classifies the AUASTs into clusters, and outputs the detailed view of structural generalization (LUS) 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.

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

	Tomcat	Hibernate	Camel	Solr
Log4j statements	1098	128	632	1484
LMs	658	81	490	818
Primitive clusters	1098	128	632	1484
Clusters	43	12	49	50
Generalized clusters	20	9	18	27
Singleton clusters	23	3	31	23
Reduction	96%	91%	92%	96%

Table 7.1: The experimental results.

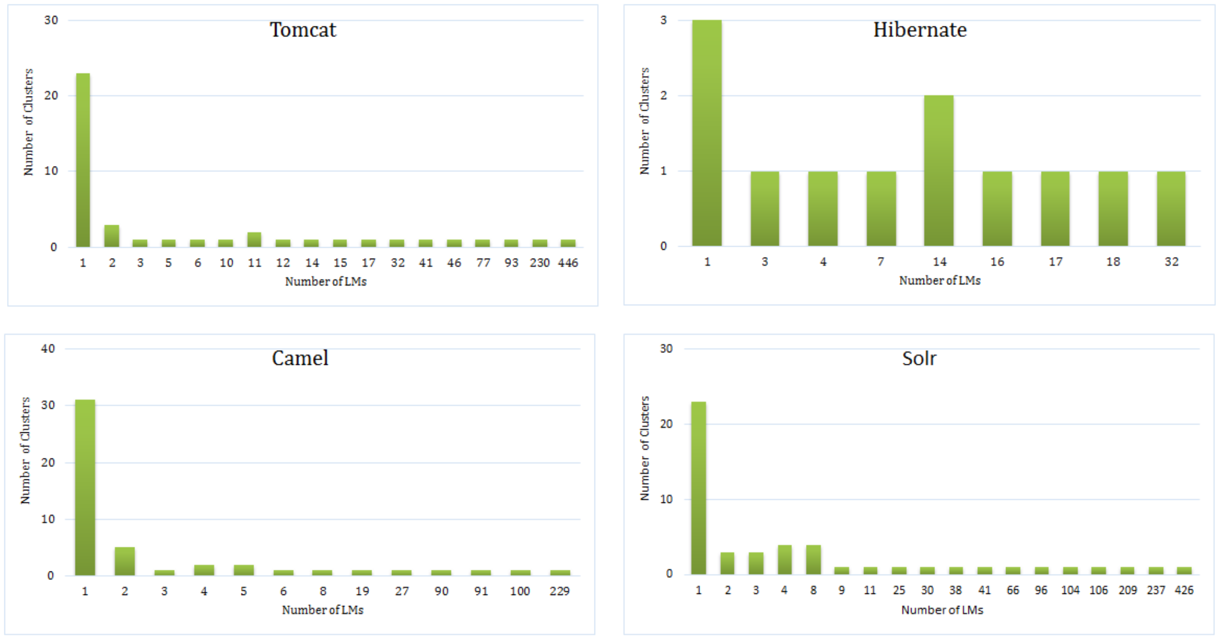


Figure 7.2: Histograms of the number of LMS per cluster.

7.1.2 Analysis

The first research question is : *"Is it possible to find patterns of where log statements occur in source code?"*. As it is 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 few number of LMs (only one or two). This indicates that in these cases developers follow a more complex or rare way of locating log statements.

The second research question is : *"What common structural characteristics do logged methods have?"* To address this question, I have manually went through the LUSs of anti-unifiers to identify the common structural characteristics of locating the 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 six categories of anti-unifiers in the logging usage. In the following sections, I will describe the common structural characteristics of each category represented by the anti-unifiers. In addition, Figure 7.3 presents the number of LMs in each category and its percentage to the total number of LMs for each of the software systems.

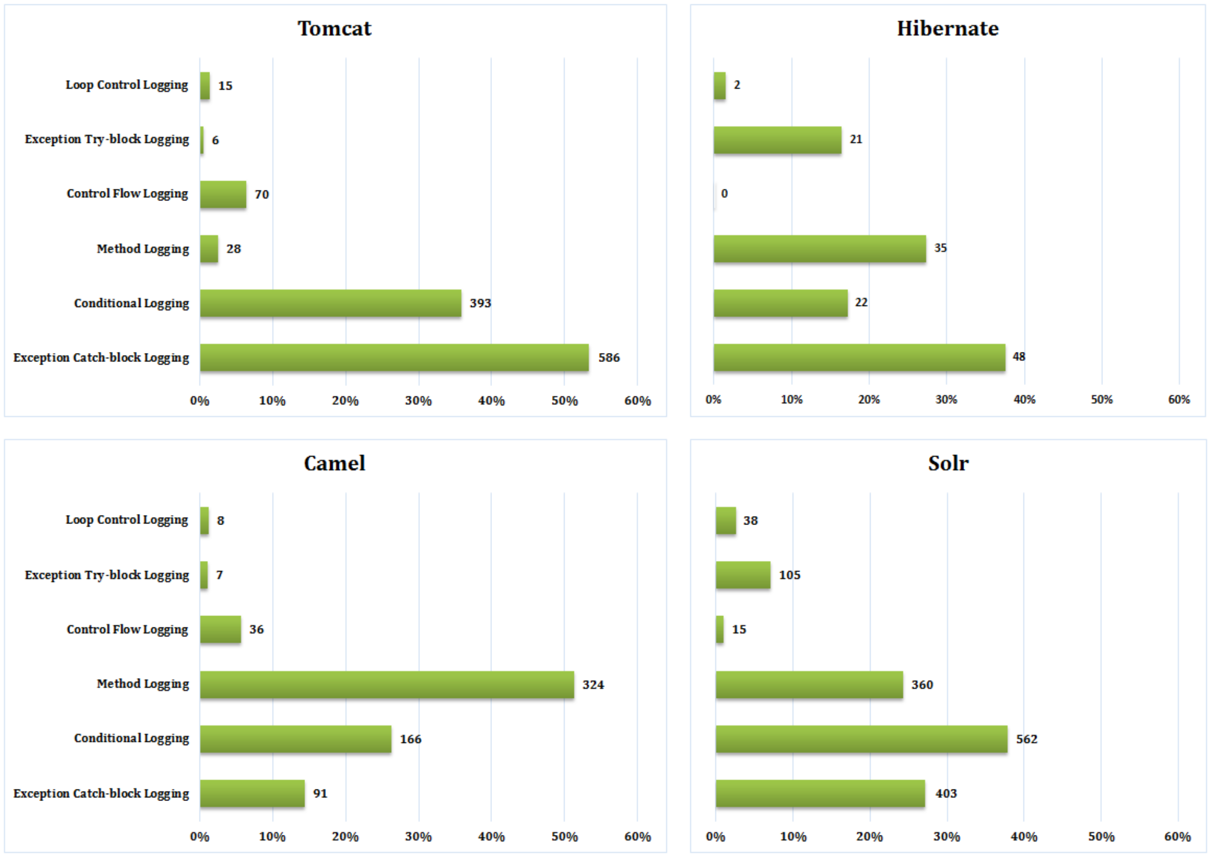


Figure 7.3: The distribution of the categories of anti-unifiers in the logging usage.

A. Exception Catch-block Logging

The main common structural characteristics of the anti-unifiers of this category are the **try** and **catch** statements, where the log statements are located inside the body of a Catch Clause. As shown in Figure 7.3, 14% to 53% 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 error handling technique in the Java programming language.

B. Conditional Logging

In this category, log statements are enclosed by **if** statements that their expressions are mostly among Infix Expression, Method Invocation, or Binary Expression nodes. The Infix Expressions mostly check if the value of a variable either equals to the Null Literal or is outside of a valid range; the **if** statements with Method Invocations mostly check if the return value of an invoked method is an indicator of a potential problem within the system; and the **if** statements with Binary Expressions mostly checks if a Boolean Literal is incorrect. As shown in Figure 7.3, 17% to 36% 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. Method Logging

In this category, the log statements are located inside the body of Method Declaration nodes. A common structural characteristic of the anti-unifiers in this category is that they mostly use the Throw Statements 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 Apache Camel. 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. Exception Try-block Logging

In this category, the log statement is located inside the body of the **try** statement of a **try/catch** block. These log statements can be used to record important information about the code that may throw an exception. According to the Figure 7.3, 1% to 16% of the total LMs of the studied systems are described by the anti-unifiers of this category.

E. Control Flow Logging

In this category, the log statements are located inside the body of either **switch**– or **if**–then–**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 6% of the total LMs are described by the anti-unifiers of this category.

F. Loop Control Logging

The log statements of this category are located inside the **for**−, enhanced−**for**−, **while**−, or **do** −**while**− statements. These log statements are mostly used to log important information about every object of a Java Collection. This information might be helpful to diagnose the root causes of a failure within a system. According to the Figure 7.3, a low percentage of LMs (1% to 3%) are described by the anti-unifiers of this category. This suggest that in practice, it is not a common way of using log statements in source code.

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 describe the process of evaluating the precision and recall of ELUS.

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 (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 five 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 the 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 are 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.

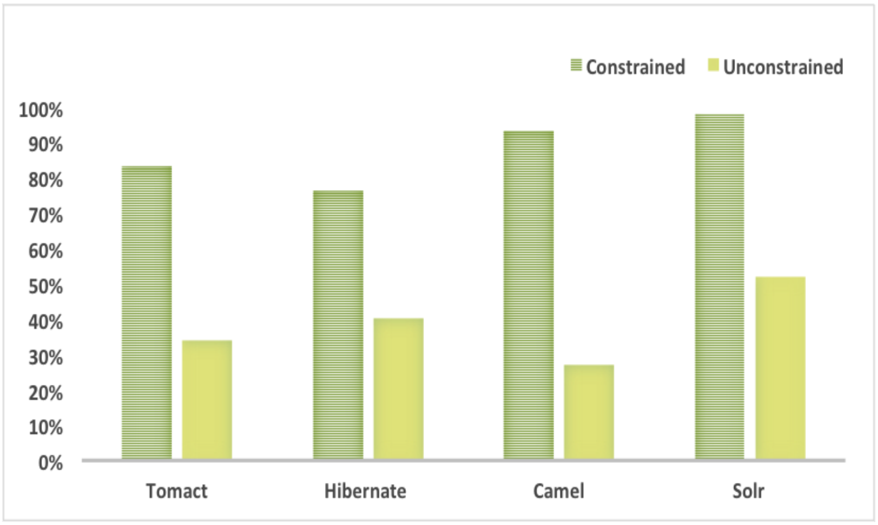


Figure 7.4: The precision of ELUS.

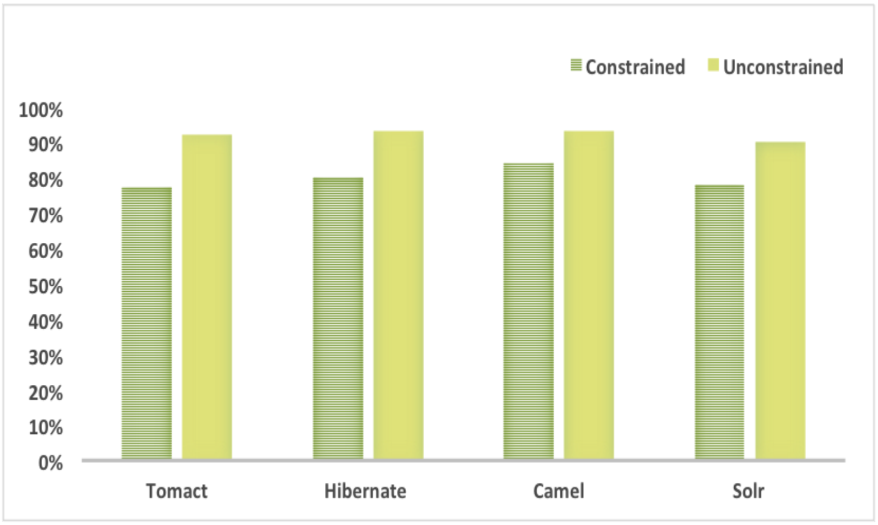


Figure 7.5: The recall of ELUS.

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 five software systems. The average precision for ELUS is 88% and 38% for constrained and unconstrained variable experiments, respectively. In general, the precision for constrained variables are fairly high. The main reason behind the high precision is that in these cases the variables can only be substituted with some particular nodes, which makes the anti-unifier very specific. Furthermore, according to the Figure 7.4, the precision is fairly low for unconstrained variables. The main reason for 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.

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 92% 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 in detecting 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 detect correct locations of log statements in the source code.

7.3 Summary

I conducted an experimental study to characterize the location of log statements by applying my tool on the source code of five 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 constructing 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 to categorize the anti-unifiers of logging usage.

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), and the usage of anti-unification theory for other applications (Section 8.3).

8.1 Threats to validity

Prior to applying our tool for characterizing logging usage in real-world software systems, I have conducted three experiments to investigate the effectiveness of the proposed approach. However, there are several potential threats regarding the validity of these experiments. First, the results of my manual examination might be biased, as I determined the correct correspondences between AUASTs and the correct way of classifying the set of AUASTs in our test suite based on a similarity measurement. To limit the bias, other people can be involved to double check the accuracy of my manual inspection in a future work. Secondly, the experiments have examined one test suite containing a set of LMs from a real-world software system, though different test suites may generate different results. Although I cannot claim that the LMs in my test suite are a good representative of all LMs in real-world software systems, the results are still promising, as logging calls are used in various ways in Java methods of my test suite, and have sufficed to indicate the effectiveness of my approach in constructing structural generalizations. Another potential thread is that the successful rate of detecting correspondences by our tool might happened accidentally only for our test suite. To resolve this doubt, I examined the cases where our 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 generalization through the use of structural correspondence. That is, our tool creates structural generalizations successfully with regard to what our algorithm should generate. A potential thread to the validity of our characterization study is the degree to which our sample set of software systems is a good representation of all real-world logging usage. To address this issue, we selected various open-source software projects in terms of application, including a programming text editor, a web server, and an application server. These software systems are among the most popular applications in their own product category, and they all have at least 10 years of history in software development. However, our findings might not be able to reflect the characteristics of logging usage in other types of systems such as commercial software systems, or software written in other programming languages.

8.2 The pitfalls of my tool

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

8.2.1 Inaccurate node ordering

Our anti-unification algorithm does not guarantee to maintain the correct sequence of statements in the body of methods when anti-unifying two method declaration nodes, since the order of statement nodes is not considered in determining the best correspondences. For example, consider we have two corresponding methods $method_1$ and $method_2$ embodying a_1, a_2, a_3 and b_1, b_2 sequences of statements, respectively. If our tool finds that the b_1 and b_2 nodes are the best correspondences for the a_3 and a_1 nodes respectively, the output generalization view for the set of statement nodes would be a_1-or-b_2, a_2-or-NIL, a_3-or-b_1. Therefore, the generalization view does not preserve the correct ordering of nodes in the original structures.

8.2.2 Conflict resolution

The decisions I have made to resolve the conflicts occurred in constructing structural generalizations 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 higher number of leaves, though it might be not 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 be corresponded. 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 me to get approximate results, they helped to limit the complexity of my approach allowing the implementation of it as a practical solution.

8.3 Applications of anti-unification

Our study demonstrates the application of an extended from of anti-unification (HOAUMT) to infer usage patters of log statements in source code via the creation of structural generalizations. Anti-unification and its extensions have been already applied to solve several theoretical and practical problems, such as analogy making [Schmidt, 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 can be used to create generalizations in different contexts, and therefore the set of equational theories should be developed particularly for the higher-order structure used 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, these theories should ensure that only a finite number of anti-instances exist for each structure. Taking all these considerations into account enables HOAUMT to anti-unify sets of structures in a particular context. The practical tests I have conducted through the application of my tool on a test suite demonstrate that our approximation of HOAUMT was successful in constructing structural generalizations required to solve our problem.

8.4 Summary

I discussed the potential threads to validity of my evaluation and characterization study. To limit the bias of the experiments I conducted to evaluate the effectiveness of my approach and the tool support, I selected the test cases form a real system with various levels of similarity in the usage of logging calls. Furthermore, I examined the failed test cases to assure that our tool works when it should work with regard to the proposed algorithm. I will also make our test suite available for public examination to further check the accuracy of our manual inspection. For the characterization study, I selected various software systems in terms of functionality that are widely used by many developers for a long period of time. 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 an approximated 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.

Chapter 9

Related Work

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

9.1 Usage of logging

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

- **Log analysis for failure diagnosis:** Xu et al. [2009] use statistical techniques to learn a decision tree based signature from 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.

- **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 causes a specific system behavior. The Linux Trace Toolkit [Yaghmour and Dagenais, 2000] was created to record and analyze system behavior by providing an efficient kernel-level event logging infrastructure. A more flexible approach is taken by DTrace [Cantrill et al., 2004] which allows dynamic modification of kernel code.

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

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

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. 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 on four open-source software systems by mining their revision history. Their study shows that developers spend great effort modifying log statements as after-thoughts, which indicates that they are not satisfied with the log quality in their first attempt. They also characterize where developers spend most of their time in modifying the log messages.

Yuan et al. [2012a] study the problem of lack of log messages for error diagnosis and suggests to log when generic error conditions happens. 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 but still needed to be logged. Moreover, these studies mainly research log message modifications and potential enhancements of them, however, the focus of this study is on understanding where logging calls are used in source code.

9.2 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 find common subtrees in the same partition through a tree comparison algorithm. Apiwattanapong et al. [2004] present a top-down approach to detect differences and correspondences between two versions of a Java program, through comparison of the control flow graphs created from source code. Holmes et al. [2005] 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 our context.

Cossette et al. [2014] present a new approach, called matching via structural generalization (MSG), to recommend replacements for API migration. They used Jigsaw to find structural correspondences, however, the proposed 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 our problem.

9.3 API usages patterns

Various data mining approaches have been used to extract API usages 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, 2008] is a tool developed to help developers understanding how to use APIs in their source code through calculating popularity statistics for each API of a library. Acharya et al. [2007] present a framework to extract API usage scenarios as partial orders, 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 detailed structural generalizations needed in our context.

9.4 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 our context. Cottrell et al. [2008] developed Jigsaw to help developers integrate small-scale reused code into their own source code by determining structural correspondences through the application of higher-order anti-unification modulo theories. Although I used the Jigsaw framework to find potential correspondences between AST nodes, their approach does not suffice to construct structural generalizations of a set of source code fragments by considering the limitations of this study in determining correspondences.

9.5 Clustering

Clustering is an unsupervised machine mining technique that aims to organize a collection of data into clusters, such that intra-cluster similarity is maximized and the inter-cluster similarity is minimized [Karypis et al., 1999, Grira et al., 2004]. We divided existing clustering approaches into two major categories: partitional clustering and hierarchical clustering. Partitional clustering try to classify a data set into k clusters such that the partition optimizes a pre-determined criterion [Karypis et al., 1999]. The most popular partitional clustering algorithm is k-means, which repeatedly assigns each data point to a cluster with the nearest centroid and computes the new cluster centroids accordingly until a pre-determined number of clusters is obtained [Bouguettaya et al., 2015]. However, the k-means clustering algorithm is not a good fit to our problem, as it requires to predefine the number of clusters we need to come up with, which is not reasonable in our context.

Hierarchical clustering algorithms produce a nested grouping of clusters, with single point clusters at the bottom and an all-inclusive cluster at the top [Karypis et al., 1999]. Agglomerative hierarchical clustering is one of the main stream clustering methods [Day and Edelsbrunner, 1984] and has applications in document retrieval [Voorhees, 1986] and information retrieval from a search engine query log [Beeferman and Berger, 2000]. It starts with singleton clusters, where each contains one data point. Then it repeatedly merges the two most similar clusters to form a bigger one until a pre-determined number of clusters is obtained or the similarity between the closest clusters becomes below a pre-determined threshold value. Hierarchical clustering algorithms work implicitly or explicitly with the $n \times n$ similarity matrix such that an element in row i and column j represents the similarity between the ith and the jth clusters [Karypis et al., 1999].

There are various versions of agglomerative hierarchical algorithms that mainly differ in how they update the similarity between clusters. There are various methods to measure the similarity between clusters, such as single linkage, complete linkage, average linkage, and centroids [Rasmussen, 1992] **[RW: Put into bibtex]** . 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 of all data points in the cluster, and the similarity between two clusters is measured by the similarity of the clusters' centroids. However, in our application, each cluster is composed of one AUAST, and the similarity between two clusters is measured by the similarity between the clusters' AUASTs, which is the ratio of the number of common pieces over the total number of pieces of their anti-unifier.

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. However, to my knowledge, no study has been conducted on characterizing where logging is used in source code via structural generalization and clustering. 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 Java methods and grouping them into clusters based on structural correspondence.

Chapter 10

Conclusion

Determining the detailed structural similarities and differences between a set of source code fragments is a complex task that can be applied to solve several source code analysis problems. As a specific application, the focus of this study is on detecting usage patterns of logging calls in source code via structural generalization and clustering.

Logging is a pervasive practice and has various applications in software development and maintenance. However, it is a challenging task for developers to understand how to use logging calls in source code. I have presented an approach to automatically characterize where logging calls happen in source code. I have developed a prototype tool implementing my proposed approach that proceeds in three steps. First, it extracts the ASTs of logged Java methods using the Eclipse JDT framework, extends the AST structures to AUAST, and determines potential structural correspondences between AUAST nodes via the Jigsaw framework. Second, it constructs an anti-unifier form AUASTs of two given LMs with a focus on logging calls through the implementation 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 logging calls with anything else. It also develops a measure of structural similarity that determines how similar is the usage of logging calls in these Java methods. Third, it classifies a set of logged Java methods via a hierarchical clustering algorithm suited to our application.

The application of HOAUMT to construct generalizations via determining structural correspondences is novel to the problem of extracting usage patterns of log statements in source code. To evaluate the effectiveness of this approach in constructing generalizations and clustering logged

Java methods, three experiments were conducted on a sample test suite. I found that my tool was successful in determining correct correspondences for my application in % of test cases. It was also successful in clustering logged Java methods of our 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.

Furthermore, an study was conducted to infer logging usage patterns of the source code of three software systems on a method-granularity basis via my tool that generates a detailed view of structural generalizations describing the commonalities and differences between the usage of logging calls in Java methods. Our characterization study shows

In summary, our study makes the following contributions:

- An approach to constructing a structural generalization from ASTs of two logged Java methods with special attention to log statements by determining structural correspondences and developing an approximation of higher-order anti-unification modulo theories for our context.
- An approach to developing a similarity measure that indicates the level of similarity between the usage of logging calls in two Java methods.
- A hierarchical clustering algorithm to clustering a set of Java methods showing different usages of logging calls into different clusters.
- An approach to detecting usage patterns of log statements in source code via structural generalization and clustering.

10.1 Future Work

Future work could be directed to address the remaining issues of this study as described below:

- **Improving logging practices:** characterizing logging usage could be a huge step towards improving logging practices through the provision of some guidelines that might help developers in making decisions about where to log. Further studies could be conducted to investigate the feasibility of predicting the location of logging calls based on the detected usage patterns. Future work can also be done to develop recommendation tool supports that not only save developers time and effort for making decisions about where to log, but also improve the quality of logging practices.

- **Further validation of this study:** The characterization study can be conducted on more software systems to further validate the findings of my study. In addition, a survey can be conducted to ask developers on the factors they consider to decide on where to log. It might also be helpful to recognize important structural and semantic information that should be taken into account for characterizing logging usage.

- **Further extensions to my approach and the tool support:** data flow analysis can be performed to detect the problems related to node ordering in the construction of anti-unifiers. This approach can also be extended to examine more advanced semantical and contextual information of source code fragments enclosing logging calls in addition to structural information. In addition, 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 for our problem. However, the complexity of applying all these extensions must be kept restricted to maintain the approach as a practical one.

- **Other applications:** any applications that are involved in the inference of structural patterns in source code even infrequently-used patterns might benefit from my tools underlying framework. Furthermore, understanding the commonalities and differences among source code fragments has application in several areas of software engineering, such as API usage pattern collation, code clone detection, recommending replacements for API migration, and merging different branches of version control systems. Our tool‘s functionality to construct a detailed view of structural generalizations of a set of source code fragments via structural correspondence and clustering could be used to improve the results of these 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 the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34. ACM, 2007.

Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, 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 19th IEEE international conference on Automated software engineering*, pages 2–13. IEEE Computer Society, 2004.

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

Doug Beeferman and Adam Berger. Agglomerative clustering of a search engine query log. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 407–416. ACM, 2000.

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

Athman Bouguettaya, Qi Yu, Xumin Liu, Xiangmin Zhou, and Andy Song. Efficient agglomerative hierarchical clustering. *Expert Systems with Applications*, 42(5):2785–2797, 2015.

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

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. arbeitspapier 1006, 1996.

Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, 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 *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (at OOPSLA)*, 1999a.

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

Bradley Cossette, Robert Walker, and Rylan Cottrell. Using structural generalization to discover replacement functionality for API evolution. 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 HE Day and Herbert Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of classification*, 1(1):7–24, 1984.

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

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

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.

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

Reid Holmes and Robert J Walker. A newbie's guide to eclipse apis. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 149–152. ACM, 2008.

Reid Holmes, Robert J Walker, and Gail C Murphy. Strathcona example recommendation tool. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 237–240. ACM, 2005.

Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding customer problem troubleshooting from storage system logs. In *FAST*, volume 9, pages 43–56, 2009.

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. In *SIGSOFT Software Engineering Notes*, volume 30, pages 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 *USENIX Annual Technical Conference*, 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 *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 353–366, 2012.

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

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

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. *Software Engineering, IEEE Transactions on*, 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 2006 international workshop on Mining software repositories*, pages 65–71. ACM, 2006.

Martin Schmidt. Restricted higher-order anti-unification for heuristic-driven theory projection. Bachelor's thesis, University of Osnabrück, Osnabrück, Austria, 2010. URL http://ikw.uni-osnabrueck.de/en/system/files/31-2010.pdf.

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 22nd symposium on Operating systems principles*, pages 117–132. ACM, 2009.

Karim Yaghmour and Michel R Dagenais. Measuringandcharacter izing systembeh av ior usingkernel-leveleventlogging. 2000.

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

Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 293–306, 2012a.

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

Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):4, 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 *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 415–425. IEEE, 2015.