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

Acknowledgements

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

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

Chapter 1

Introduction

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

Logging is a conventional programming practice of recording an application's state and/or actions during the program's execution [Gupta, 2005], and log system analysis assist developers in diagnosing the presence or absence of a particular event, understanding the state of an application, and following a program's execution flow to find the root causes of an error. The importance of logging is notable in its various applications in software development and maintenance tasks 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].

In practice, logging tasks can be performed in various ways, as developers may make different decisions about where and what to log. For example, they can apply logging to record the occurrence of every event of an application and so, they use logging calls at the start and end of the body of every method in the source code [Clarke et al., 1999a,b]. However, three main problems are associated with excessive logging. First, it can generate redundant information that might be

confusing and misleading for developers performing system log analysis, as it masks significant information. 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 logging 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.

Despite the importance of logging for software development and maintenance, few studies have been conducted on understanding logging usage in real-world applications, since logging has been considered to be a trivial task [Clarke et al., 1999a,b]. However, the availability of several complex frameworks (e.g., 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 efficiently on their first attempt.

Research on the problem of characterizing logging practices can be divided into two main topics: context and location of logging calls. The context refers to the log text messages and the location refers to where logging calls are used in the source code. A few studies have been conducted on characterizing log message modifications [Yuan et al., 2012b] and developing tools to automatically enhance the context of existing logging calls [Yuan et al., 2012c, 2010]. However, no research has been conducted on studying the location of logging calls in real-world software systems, though it has a great impact on the quality of logging as it helps developers to trace the code execution path to identify the root causes of an error for log system analysis. Through this research, I address this gap by developing an automated approach to detecting the detailed structural similarities and differences in the usage of logging, both within a system and between systems.

1.1 Programmatic support for logging

A typical logging call 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 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: Logging call examples from the log4j framework.

1.2 Broad thesis overview

I aim to create an approach that provides a concise description of where logging is used in the source code by constructing generalizations that represent the detailed structural similarities and differences between entities that make logging calls, 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; my study investigates the location of logging calls from the point of view of logged methods. 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 logging calls. Therefore, my approach must take the logging calls into account when I perform the generalization task via the determination of structural correspondences.

[RW: Your research is not about implementing a tool. Your research is about developing an approach, a concept, which you then implement in order to perform experiments. The concept does not depend on Java, or the JDT, or Jigsaw: these are implementation-level choices. If your implementation contains bugs, this does not immediately reflect on the concept. If the concept has bugs, this will be reflected in the implementation.] 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 applies some constraints prior to determining the best correspondences, in order to prevent the anti-unification of logging calls with any other type of nodes in the tree structure. The anti-unifier is constructed through the anti-unification of each AUAST node with its best correspondences and then a mea-

sure of structural similarity is developed between the two AUASTs. **[RW: Is there no separate chapter on this?]**

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. This step is realized by my clustering tool (Chapter 6.1).

The last step involves analyzing the structural generalizations to extract logging usage schemas that represent the structural commonalities of the set of LJMs within each cluster.

To evaluate the approach, I implemented it in a tool written in the Java programming language. I use the Eclipse JDT framework to extract the AST of LMs from a Java program. My correspondence tool (Section 4.1) is employed to create AUASTs, and my anti-unifier tool (built atop the Jigsaw framework) is applied to find potential correspondence connections between AUAST nodes (Section 5.1).

1.3 Thesis statement

The thesis of this work is that the detailed structural similarities and differences between source code that makes use of logging calls can be determined via higher-order anti-unification modulo theories, providing a concise and accurate description of where logging calls do occur in real-world software systems.

1.4 Thesis organization

The remainder of the thesis is organized as follows.

Chapter 2 motivates the problem of understanding where to use logging calls 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; anti-unification, which is a theoretical approach for determining similarities and differences in tree structures; the Eclipse JDT, an industrial framework for producing and manipulating ASTs for source code written in the Java programming language; and on Jigsaw, a research tool based on the Eclipse JDT for performing anti-unification.

[RW: This chapter needs to be clearer about what is new and what is not. By moving all background to the earlier chapter, this is better but not quite good enough. You can then have this chapter be explicit about what you added and you can have the reader be less lost in the details. Currently, you start the chapter with an algorithm that they cannot possibly understand because you have not yet explained the background. That's bad. With the structure I specified above, the background will all be in the previous chapter. Now, the reader will be able to appreciate what you are trying to do. The following description will need to be modified.] Chapter 4 describes the approach I used to determine structural correspondences and its supporting tool. In this Chapter, I provide background information on the Eclipse JDT as a framework to access and manipulate Java program via ASTs, and the Jigsaw as an existing framework to find potential structural correspondences. This Chapter also describes my experimental study on a set of LJMs selected form a real software system to validate the effectiveness of Jigsaw for my application.

Chapter 5 presents my anti-unification approach as a set of novel algorithms, and its implementation as a plug-in to the Eclipse integrated development environment (IDE). Furthermore, it describes my experimental study to assess the effectiveness of my approach and the tool support in constructing structural generalizations and computing similarity between structures. Chapter 5 describes the clustering algorithm, its tool support, and an experimental study I conducted to assess its effectiveness. **[RW: This paragraph needs modification.]**

Chapter 7 presents an empirical study I conducted to characterize the location of logging usage

in three open-source software systems. Chapter 8 discusses the results and findings of my work, threats to its validity, and the 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.

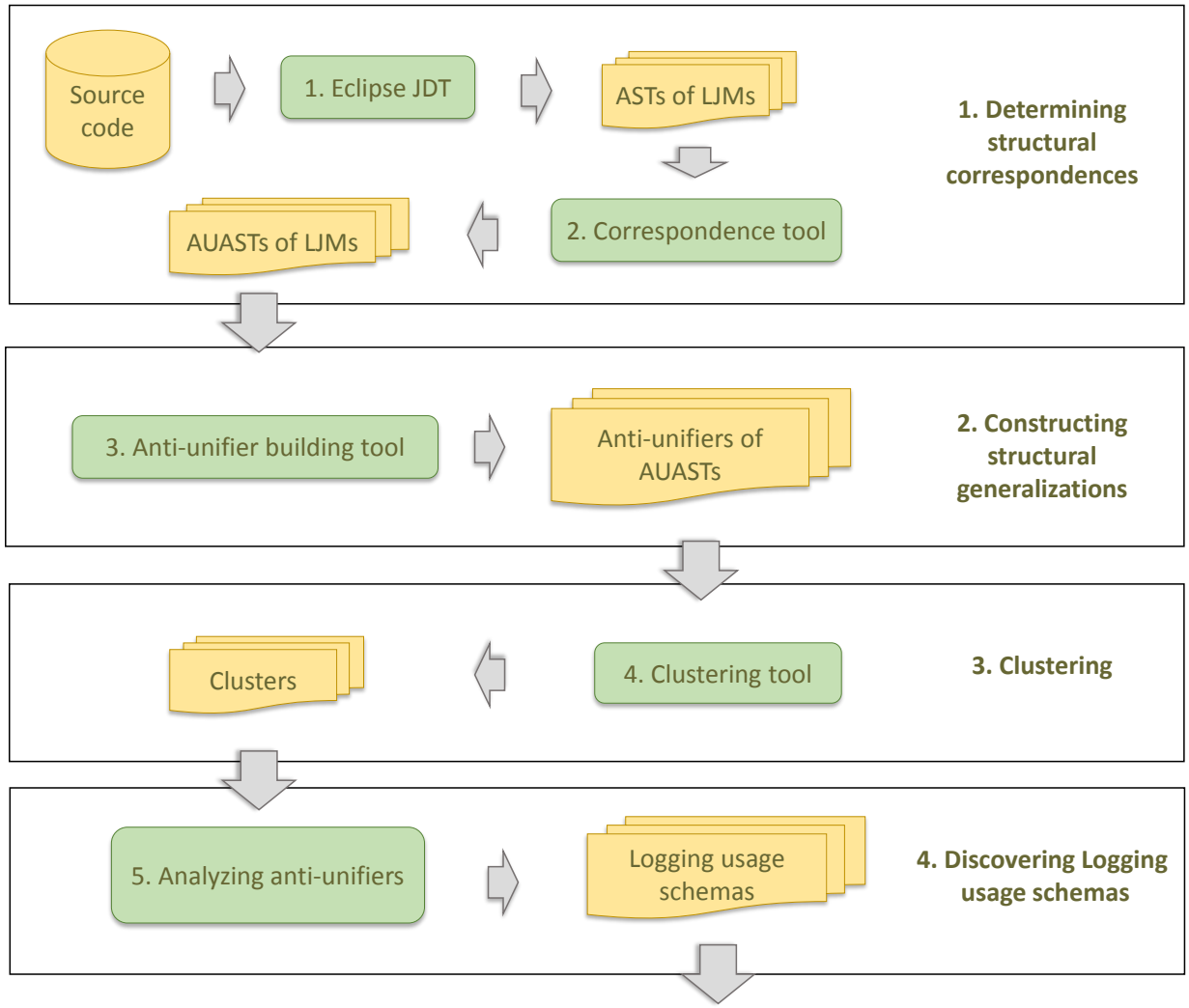


Figure 1.2: Overview of the approach. **[RW: I don't want to see mention of tools here. This should be focused on the concepts.]**

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-

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

Figure 2.1: The EditBus class.

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

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

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

Figure 2.3: The 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.

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

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

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

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

```
1  public class EditBus {
2      private static ArrayList components = new ArrayList();
3      private static EBComponent[] copyComponents;
4
5      private EditBus() {
6      }
7
8      public static void addToBus(EBComponent comp) {
9          synchronized(components) {
10             components.add(comp);
11             copyComponents = null;
12         }
13     }
14
15     public static void removeFromBus(EBComponent comp) {
16         synchronized(components) {
17             components.remove(comp);
18             copyComponents = null;
19         }
20     }
21
22     public static EBComponent[] getComponents() {
23         synchronized(components) {
24             if (copyComponents == null) {
25                 EBComponent[] arr = new EBComponent[components.size()];
26                 copyComponents = (EBComponent[])components.toArray(arr);
27             }
28         }
29         return copyComponents;
30     }
31
32     public static void send(EBMessage message) {
33         // logging call
34         EBComponent[] comps = getComponents();
35         for(int i = 0; i < comps.length; i++) {
36             try {
37                 EBComponent comp = comps[i];
38                 long start = System.currentTimeMillis();
39                 comp.handleMessage(message);
40                 long time = (System.currentTimeMillis() - start);
41                 if (time != 0) {
42                     // logging call
43                 }
44             } catch(Throwable t) {
45                 // logging call
46             }
47         }
48     }
49 }
```

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. Anti-unification is summarized in Section 3.2, 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. 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 3.4. 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

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello world!");
4     }
5 }
```

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 the concepts of "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 typed 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 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; however, non-normal programs can occur and a pure syntactic analysis cannot rule out that System is a package and that out is a class therein declaring a static method

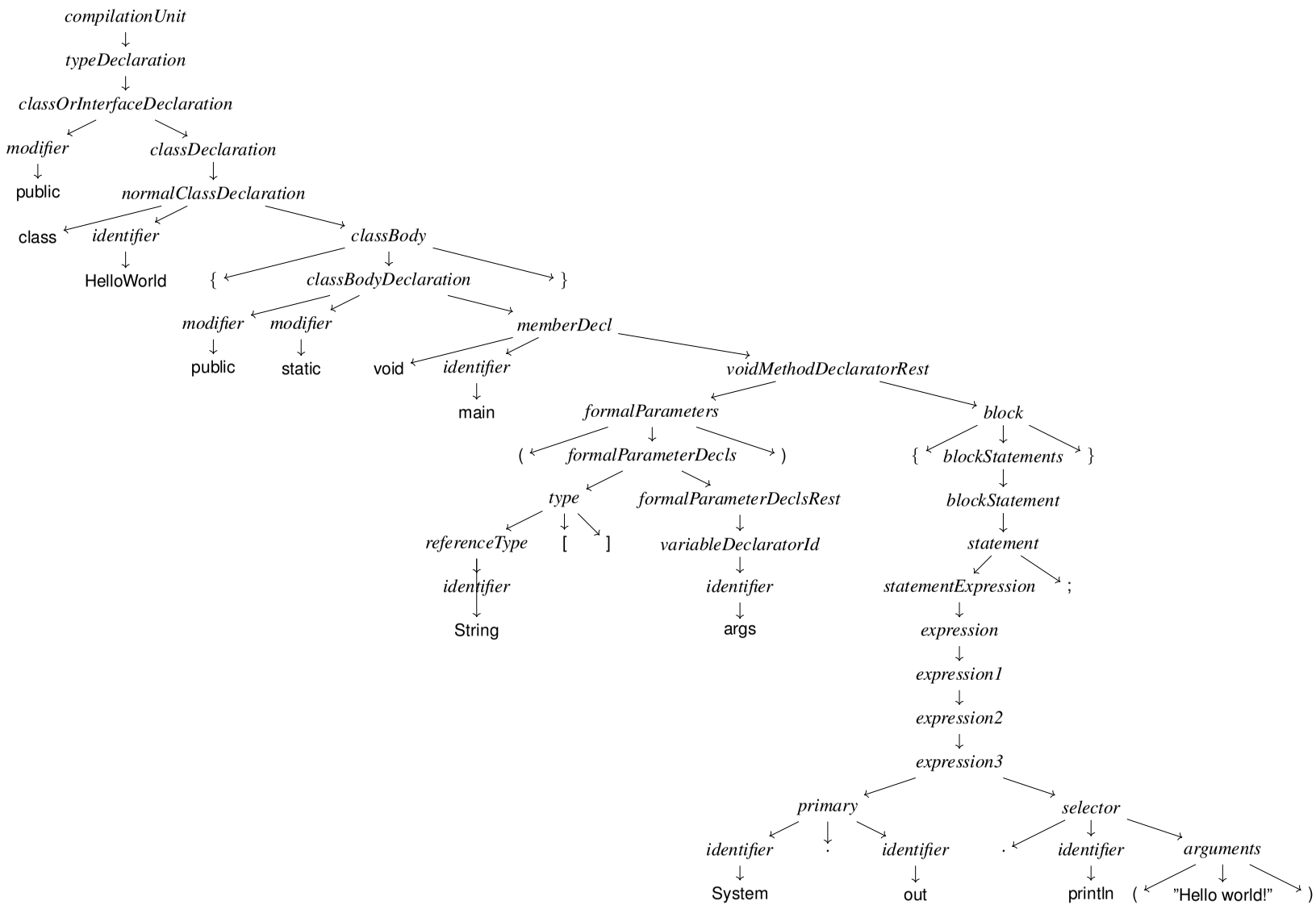


Figure 3.2: The concrete syntax tree for the program of Figure 3.1.

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 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 3.2.1 (Term)**.** A 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 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:

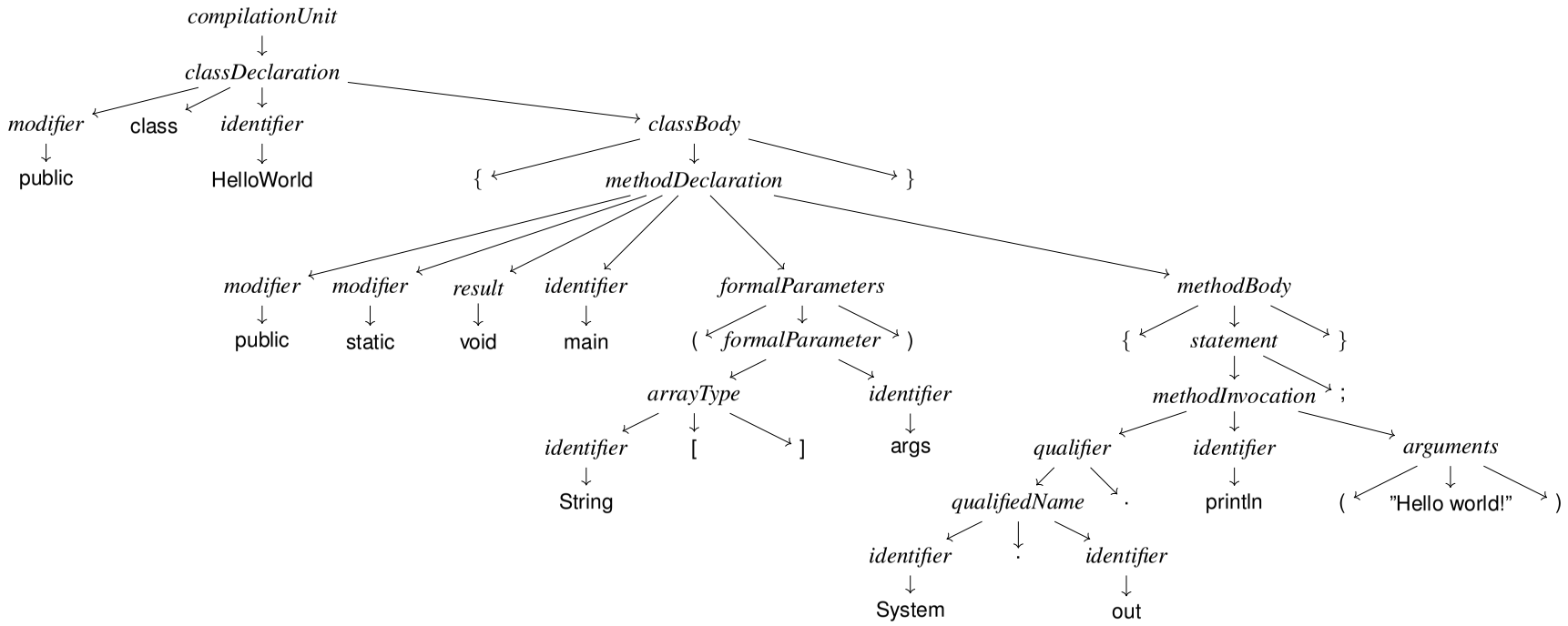


Figure 3.3: The abstract syntax tree derived from the concrete syntax tree of Figure 3.2.

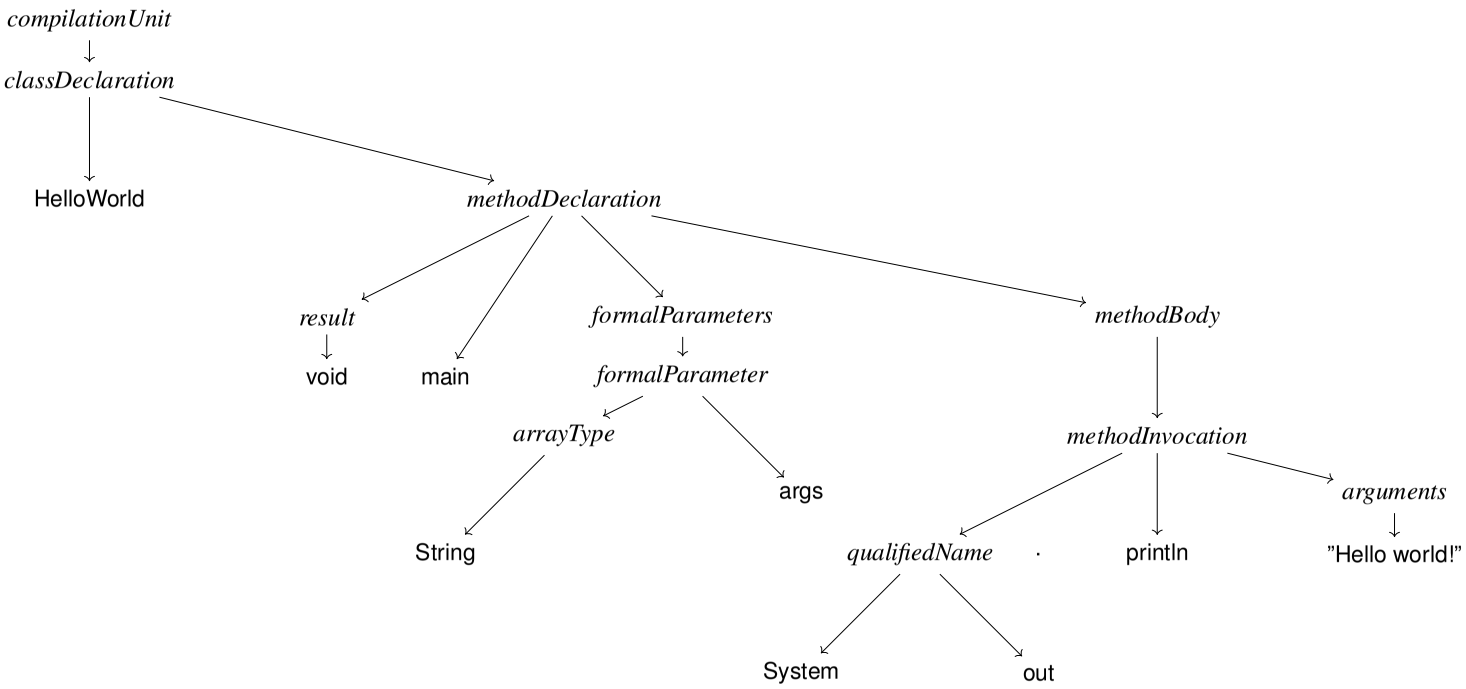
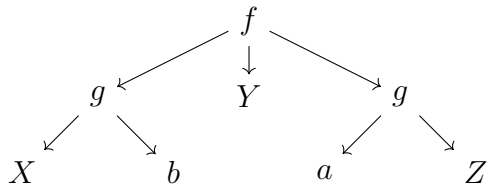


Figure 3.4: A more abstract AST derived from the concrete syntax tree of Figure 3.2.

- Y
- a
- $X \leftarrow f \rightarrow c$
- 

Definition 3.2.2 (Substitution)**.** A substitution is a set of mappings, each from a variable to a term. **[RW: Note that this definition will not work for higher-order AU.]**

Definition 3.2.3 (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 3.2.4 (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 the application of Θ on X results in a ($X \xrightarrow{\Theta} a$).

Definition 3.2.5 (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 3.2.6 (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 ($X \xrightarrow{\Theta_1} a$ and $X \xrightarrow{\Theta_2} b$).

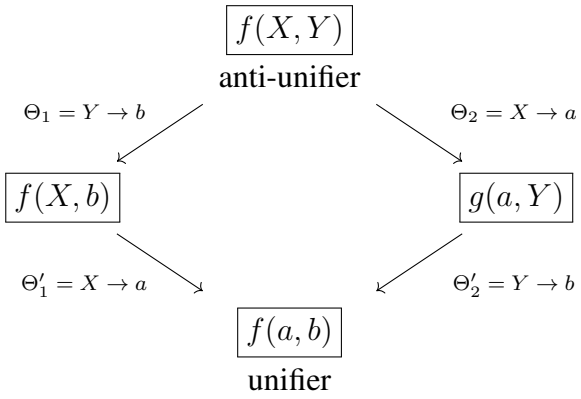


Figure 3.5: Unification and anti-unification of the terms $f(X,b)$ and $f(a,Y)$.

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 $f(X,Y)$ can be substituted by the term b to re-create $f(X,b)$ (with $\Theta_1 = Y \to b$) 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 3.5. 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 3.6.

Higher-order anti-unification would allow us to create the MSA by extending the set of possible

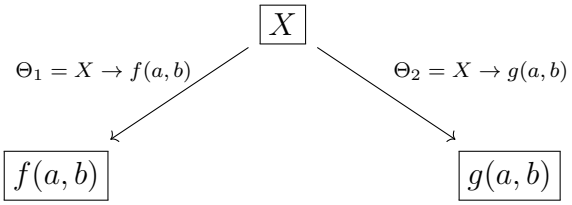


Figure 3.6: First-order anti-unification of the terms $f(a, b)$ and $g(a, b)$.

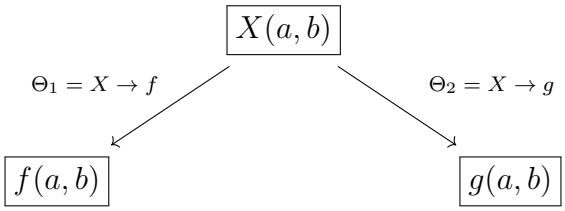


Figure 3.7: Higher-order anti-unification of the terms $f(a, b)$ and $g(a, b)$.

substitutions such that variables can be replaced not only by terms but also by function symbols in 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 3.7.

3.3 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

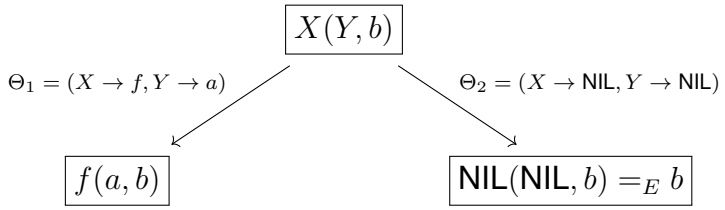


Figure 3.8: Higher-order anti-unification modulo theories of the terms $f(a, b)$ and b.

size for dissimilar substructures. For example, we can anti-unify the two structures b and $f(a, b)$ 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 3.8.

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 3.9.

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 3.10, 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

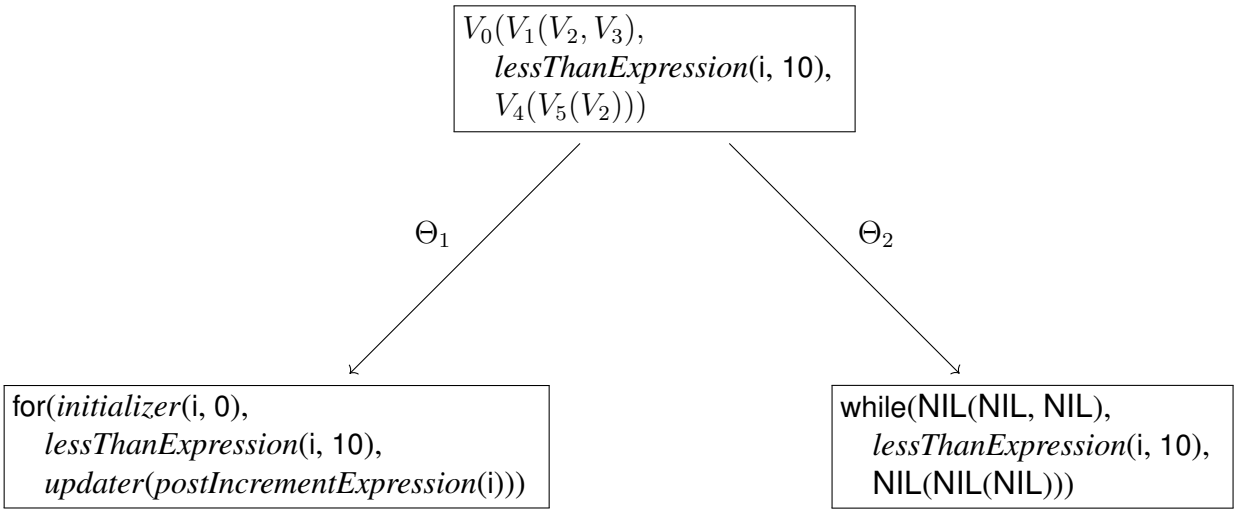


Figure 3.9: Anti-unification of the structures **for**(*initializer*(i, 0), *lessThanExpression*(i, 10), *updater*(*postIncrementExpression*(i))) and **while**(NIL(NIL, NIL), *lessThanExpression*(i, 10), NIL(NIL, NIL)). The substitutions are defined as follows: $\Theta_1 = (V_0 \rightarrow \text{for}, V_1 \rightarrow \textit{initializer}, V_2 \rightarrow \text{i}, V_3 \rightarrow 0, V_4 \rightarrow \textit{updater}, V_5 \rightarrow \textit{postIncrementExpression})$; and $\Theta_2 = (V_0 \rightarrow \text{while}, V_1 \rightarrow \text{NIL}, V_2 \rightarrow \text{NIL}, V_3 \rightarrow \text{NIL}, V_4 \rightarrow \text{NIL}, V_5 \rightarrow \text{NIL})$

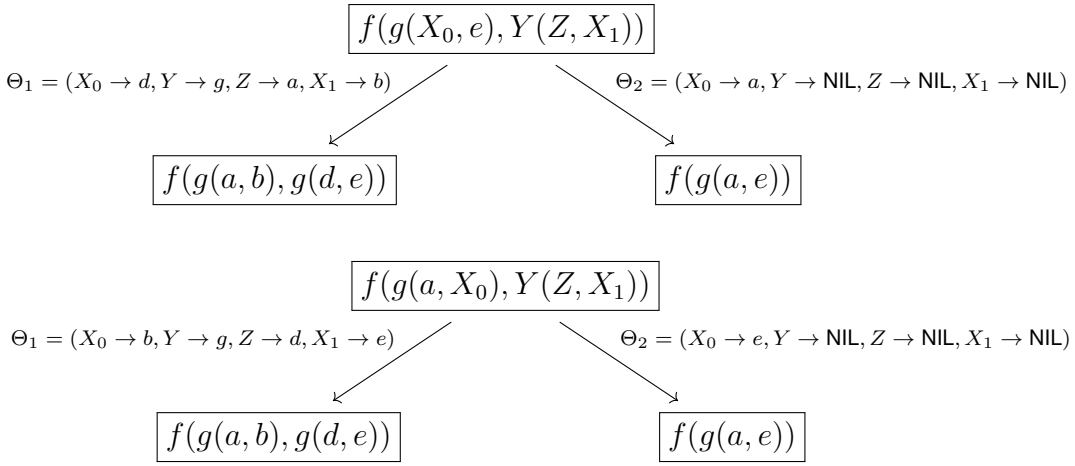


Figure 3.10: Ambiguous higher-order anti-unification modulo theories of the terms $f(g(a, b), g(d, e))$ and $f(g(a, e))$, creating multiple MSAs.

the anti-unifier $Y(Z, X_1)$; or we can anti-unify $g(a, e)$ and $g(d, e)$ to create the anti-unifier $g(X_0, e)$ 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.

3.4 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. Thus, any Java source code can be represented as a tree of AST nodes. For example, the simple AST structure of two sample LMs in Figures 3.11 an 3.12 is shown in Figure 3.13.

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

```
1 public void handleMessage(EBMessage message) {
2         if (seenWarning) return;
3         seenWarning = true;
4         Log.log(Log.WARNING, this, getClassName() + " should extend EditPlugin not
              EBPlugin since it has an empty " + handleMessage());
5     }
```

Figure 3.11: A Java method that uses a logging call. This will be referred to as Example 1.

```
1 public void actionPerformed(ActionEvent evt) {
2         EditAction action = context.getAction(actionName);
3         if (action == null) {
4                 Log.log(Log.ERROR, this, "Unknown action: " + actionName);
5         }
6         else
7                 context.invokeAction(evt, action);
8         }
```

Figure 3.12: A Java method that uses a logging call. This will be referred to as Example 2.

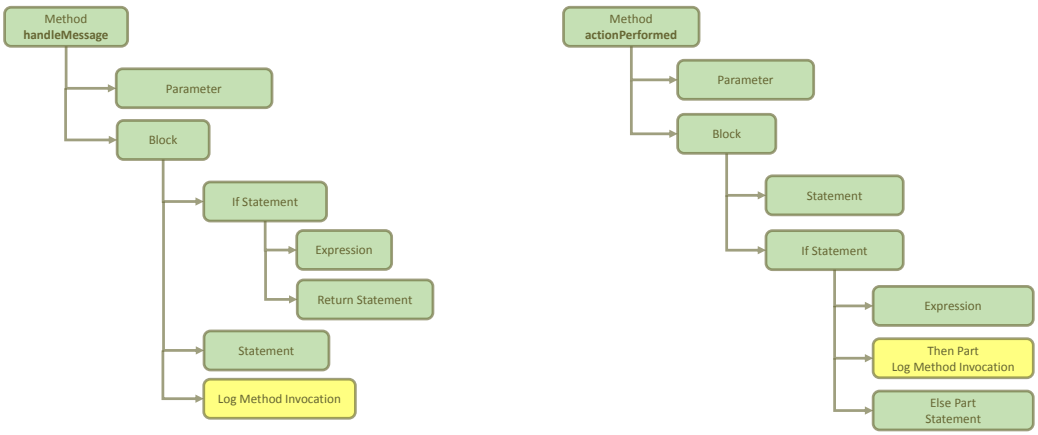


Figure 3.13: Simple AST structure of the examples in Figures 3.11 and 3.12.

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; this would be a child structural property.

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

As an example, the ASTs of the logging calls at line 4 of Figure 3.11 and Figure 3.12 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))))

3.5 The Jigsaw framework

The Jigsaw tool was developed by Cottrell et al. [2008] to support small-scale source code reuse via structural correspondence. Small-scale reuse task can be divided into two phases. The first phase involves the developer identifying a source code snippet that implements a functionality that is missing within a target system. The second phase involves modifying the source code snippet to fit in the target system. To perform a reuse task, developers mostly copy and paste the reused code snippet and then fit it to the hole of the target context by performing some modifications. Jigsaw support the small scale reuse task by identifying structural correspondences between the copied code snippet and the context in which the code wants to be pasted into, 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. Jigsaw 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.

Jigsaw generates an augmented form of AST, called a *correspondence AST* (CAST), where each node holds a list of candidate correspondence connections between the nodes of two AUASTs, each implicitly representing an anti-unifier. Jigsaw also provides a measure of structural similarity to indicate how similar the nodes involved in each correspondence connection are. The Jigsaw similarity function relies on structural correspondence along with simple knowledge of semantic equivalences supported by the Java language specification. It returns a value in $[0, 1]$ where zero indicates complete lack of similarity and one indicates perfect similarity. In addition, several semantical heuristics are used to improve the accuracy of similarity measurement by allowing the comparison of AST nodes that are not syntactically identical but are semantically related to

each other.

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 **int** and **long** types is another example, where an arbitrary value of 0.5 is defined as the similarity value, since they are not syntactically identical but are semantically related. In addition, the Jigsaw framework also detects the structural correspondence between **for**-, enhanced−**for**-, **while**-, and **do**-loop statements; and **if** and **switch** conditional statements. As an example, Figure 3.14 shows the structural correspondence connections created by Jigsaw between the AUAST nodes of Examples 1 and 2.

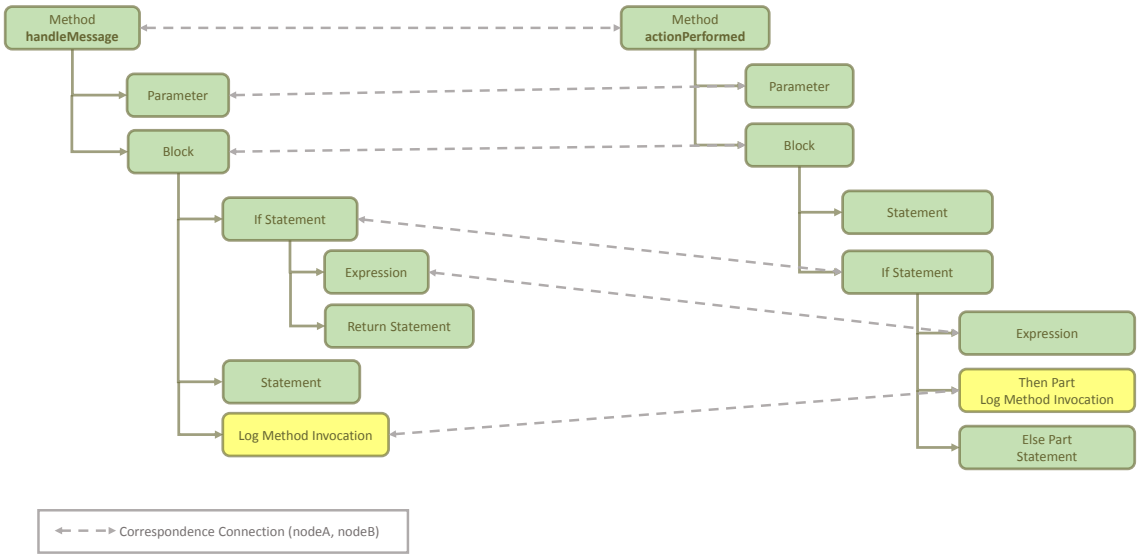


Figure 3.14: Simple CAST structure of the examples in Figures 3.11 and 3.12. The links between AST nodes indicate structural correspondence connections created by the Jigsaw framework.

3.6 Summary

We described abstract syntax trees (ASTs) as a standard syntactic representation of source code. Every AST can also be represented in a function format (and vice versa) which constitute the standard theoretical concept of terms. We demonstrated how the theoretical framework of anti-unification 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 permit 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 higher-order anti-unification modulo theories approach 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 weakness, we must apply an approximation technique to select amongst the best MSAs in order to reach a solution that is reasonable in practice.

[RW: Expand to cover new sections.]

Chapter 4

Determining Structural Correspondences

In order to construct structural generalizations describing the commonalities and differences between logged methods (LMs), we need to find structural correspondences between their ASTs. Therefore, we first need a concrete framework for constructing and manipulating ASTs. The Eclipse integrated development environment provides such a framework in its Java Development Tools (JDT) component. The details of our implementation will depend on certain details of Eclipse JDT, so we describe those in Section 3.4.

The correspondence tool, is an Eclipse plug-in I used to determine structural correspondences between nodes in the tree structures. As AST structure does not allows the insertion of variables, I first created an extended form of AST, called AUAST (Anti-unifier AST) that addresses this issue (see Section 4.1.1), and then applied Jigsaw [Cottrell et al., 2008] on these structures, which is a framework exists for determining structural correspondences and measuring similarity between the nodes involved in each correspondence (described in Section 3.5). We build atop that work in order to create our anti-unifiers. In addition, an experimental study was conducted to validate the effectiveness of Jigsaw for our application.

4.1 The Correspondence tool

The correspondence tool is an Eclipse plug-in that inputs ASTs of LJMs of a Java program, creates an extended form of AST structure, called AUAST, to allow the application of HOAUMT, and determines structural correspondences between AUASTs using the Jigsaw framework in a pairwise manner. This process will be described in the following sections.

4.1.1 Constructing the AUAST

[RW: This is a concrete implementation, not a generic idea, at least not the way it is described. I strongly suggest that you give a generic description of your assumptions about ASTs then relate AUASTs to those, then talk about implementation details.] [NZ: I EXPLAINED IT AS A PART OF THE IMPLEMENTATION OF MY TOOL.]

The goal of this phase is to construct an extended form of AST structure that would allow the creation of an anti-unified structure. As described in Section 3.3, an anti-unified structure utilizes variables that must be substituted with proper substructures to regain original structures. However, an AST structure does not contain any variables, and therefore an extended form of it is required, namely the AUAST (anti-unifier AST), 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. The AUAST structure addresses the limitations of AST to construct an anti-unifier by adding 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.

To provide an example to demonstrate this structure, the anti-unified AUAST constructed from the AUASTs of logging calls in Figures 3.11 and 3.12 is depicted in Figure 4.1. The structural variables X and Y are used to indicate the structural variations, where the X structural variable refers to two simple values and the Y structural variable refers to two child nodes. The substitutions are defined in Equations 4.1 and 4.2.

$$\Theta_1 = (X \rightarrow \text{WARNING}, Y \rightarrow \textit{additionExpression}(\textit{methodCall}(\textit{simpleName}(\text{getClassName}), \textit{arguments}()), \textit{stringLiteral}(\text{"should extend ..."}), \textit{methodCall}(\textit{simpleName}(\text{handleMessage}), \textit{arguments}())))) \quad (4.1)$$

$$\Theta_2 = (X \rightarrow \text{ERROR}, Y \rightarrow \textit{additionExpression}(\textit{stringLiteral}(\text{"Unknown action: "}), \textit{simpleName}(\text{actionName}))) \quad (4.2)$$

4.1.2 Determining structural correspondences

Our correspondence tool applies a part of the implementation of Jigsaw to identify structural correspondences between AUASTs and reuses the Jigsaw similarity function to measure the similarity between nodes involved in each correspondence connection. However, the Jigsaw tool does not suffice to construct an anti-unifier that is the best fit to our application, as the problem of this study is different. In addition, the Jigsaw similarity function does not measure the similarity of two LJMs with a focus on logging calls, which is needed in our context. To address these issues, we should develop a greedy selection algorithm to approximate the best anti-unifier for our problem by determining the best correspondence for each node. In the following chapter, we will discuss our approach to construct structural generalization and our implementation by means of the higher-order anti-unification modulo theories and the Jigsaw framework.

4.2 An Assessment of the Correspondence tool

[RW: Describe here the procedure you used to select the examples, etc., how you tested Jigsaw, and what your findings were. At some point, you complained that Cottrell had not done

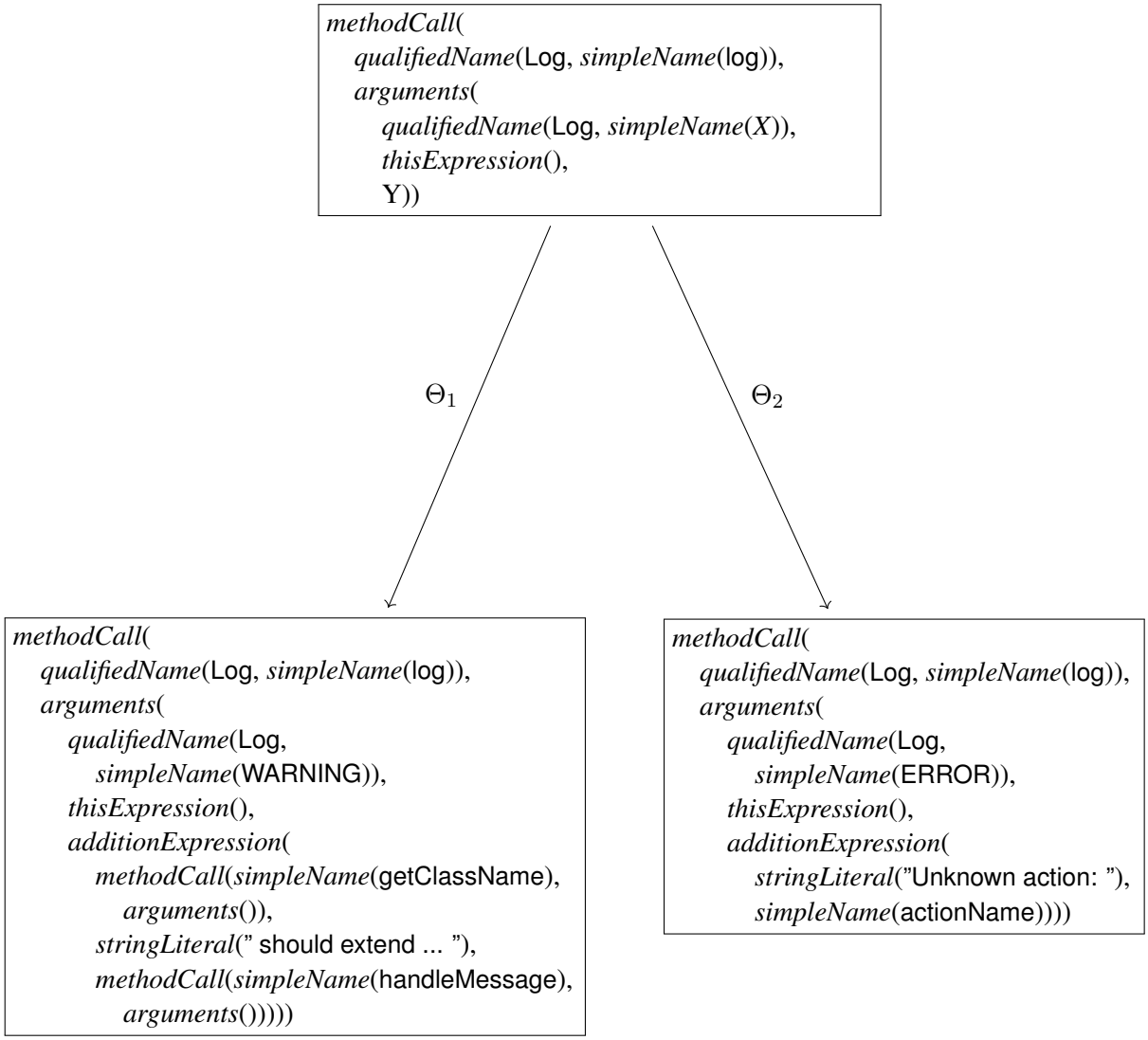


Figure 4.1: Anti-unification of the AUASTs of the logging calls in Examples 1 and 2.

something right ... do you have any evidence to demonstrate it? How does this affect your work? Such points can go in a discussion section towards the end of this chapter if they don't fit otherwise. Full details of examples can go in an appendix; here, just describe enough so people can get the point.]

[NZ: I THINK THIS EXPERIMENT IS NOT MUCH ABOUT EVALUATING JIGSAW (THE EVALUATION WAS CONDUCTED BY COTTRELL), BUT IT IS MORE ABOUT UNDERSTANDING WHAT JIGSAW DOES AND HOW TO USE IT FOR MY APPLICATION . WHAT I HAVE MENTIONED WAS ABOUT DETECTING RELEVANCE LINKS WHICH IS NOT RELATED TO MY WORK. DURING MY DEVELOPMENT, I ADDED SOME STATEMENTS TO JIGSAW FOR THE CASES THAT WAS NOT COVERED IN HIS WORK COMPLETELY (E.G, JIGSAW DID NOT DETECT THE CORRESPONDENCE BETWEEN INNER TYPE DECLARATIONS OF TWO NESTED TYPE DECLARATIONS WHEN COMPARING THE TWO UPPER TYPE DECLARATIONS]

We have conducted an experiment on a set of LJMs extracted from a real-world software system to assess how Jigsaw could effectively help us determine potential correspondences between AST nodes and measure similarity between them. We implemented a plug-in to the Eclipse integrated development environment (IDE), which uses the JDT framework to extract ASTs of a pair of LJMs and applies the Jigsaw framework to generate correspondence connections between AST nodes.

4.2.1 Setup

As a subject for our study, we used jEdit, a programmers text editor tool written in Java programming language. We chose this subject because it is a real program that has been used constantly by many developers, and it employs real usage of logging calls. Our tool extracts all LJMs within the source code of this program. However, a subset of them was selected containing 9 LJMs that showed varying levels of similarity on manual examination, and it has been used as a test suite throughout this study (see Table 4.2). The org.gjt.sp.jedit.EditBus.send(...) method contains two logging calls. To handle this case we 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. We will

describe our approach for LJMs containing multiple logging calls in details in Section 5.1.5. The last three LJMs 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.

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

Figure 4.2: Logged Java methods used as our test suite; all are contained in the org.gjt.sp.jedit package.

Our tool was used to compare LJMs of our test suite in a pairwise manner (55 test cases in total, including self-comparisons) and to produce the CASTs of each pair. The Jigsaw similarity was also measured for each of these test cases. We examined the generated CASTs of these test cases and selected a subset of 4 cases with various levels of correspondences as depicted in the Table 4.4. Case 1 contains the comparison of a Java element with itself. Case 2 contains the comparison of two Java elements that are both syntactically and semantically dissimilar. Case 3 contains the correspondence between two Java elements that are syntactically dissimilar but are

semantically relevant. Case 4 contains the comparison of a logging call with another Java element that is not logging call but is syntactically relevant.

4.2.2 Results

The results of the pairwise comparison between LJMs of the test suite is visualized in Figure 4.3. As it is shown, the Jigsaw similarity for all self-comparisons is 1, while the level of Jigsaw similarity is different for pairs containing distinct LJMs as our manual examination.

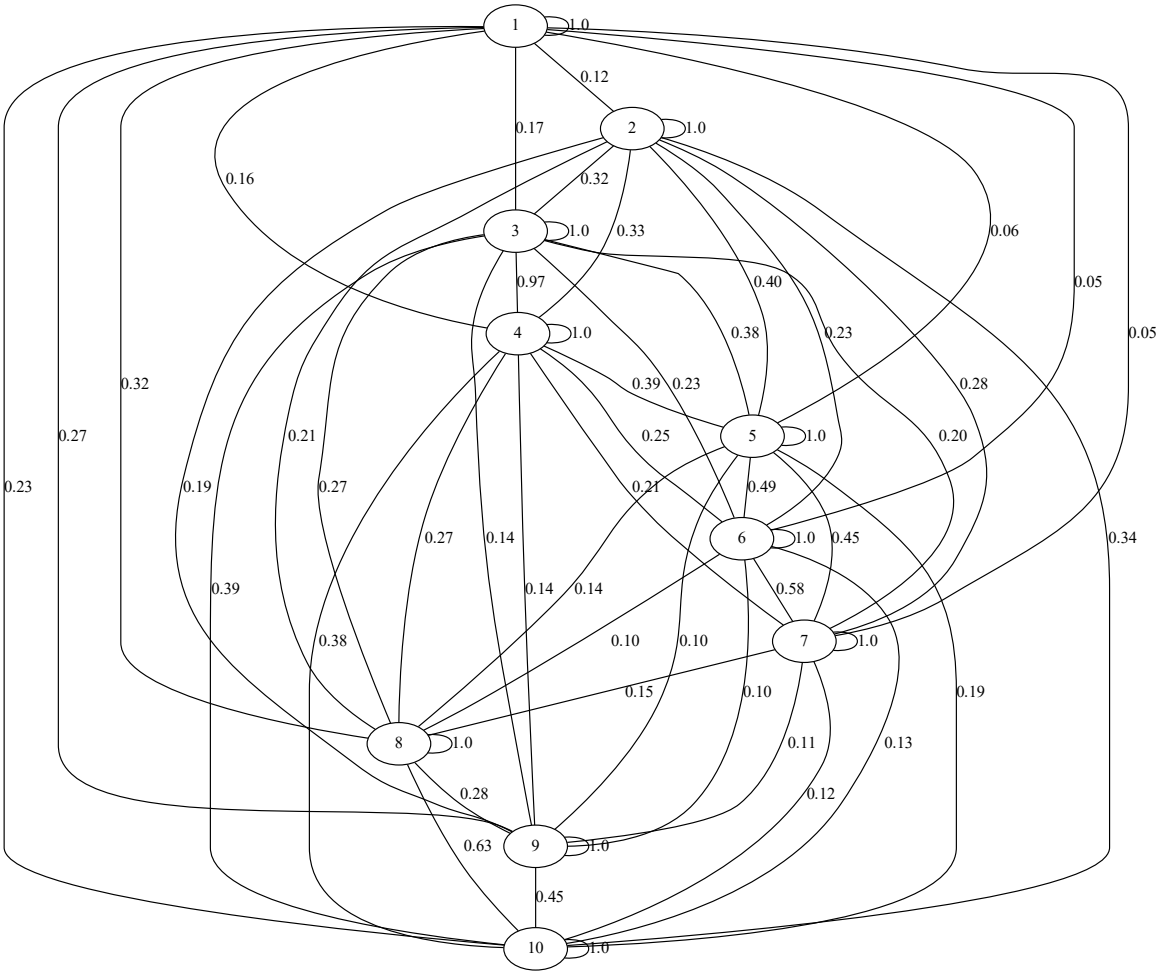


Figure 4.3: A similarity graph representing pairwise Jigsaw similarities between LJMs shown in Table 4.2.

Test case	Java source code fragment	Jigsaw similarity
1	Log.log(Log.WARNING,this,"Unknown action: " + actionName);	1
	Log.log(Log.WARNING,this,"Unknown action: " + actionName);	
2	return entry	No correspondence connection
	int i=0;	
3	for (int i=0; i ¡ comps.length; i++) ...	
	while (entries.hasMoreElements()) ...	
4	Log.log(Log.WARNING,this,"Unknown action: " + actionName);	0.33
	EditBus.removeFromBus(this);	

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

The analysis of the 4 test cases is shown in Table 4.4. Test Case 1 shows that a Java element that is compared with itself has a Jigsaw similarity of 1. Test Case 2 indicates that no correspondence connection is created when a Java element is compared with another Java element that is utterly dissimilar. Test 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. Test 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 5.1.1.

4.3 Summary

We described Eclipse JDT as a concrete framework that can be used to manipulate ASTs of a source code written in the Java programming language. We also introduced Jigsaw, an existing framework for determining structural correspondences between AST nodes and measuring similarity between them. Furthermore, we assess the Jigsaw functionality to address our problem through an empirical study on a set of LJMs selected from a real- world software system.

Chapter 5

Constructing Structural Generalizations

[NZ: I WOULD BE GRATEFUL IF YOU COULD HELP ME TO CHOOSE AN APPROPRIATE TITLE FOR THIS CHAPTER. IT DESCRIBES MY APPROACH TO CONSTRUCT ANTI-UNIFIER, ITS IMPLEMENTATION, AND THE EXPERIMENT I CONDUCTED TO EVALUATE IT.]

[RW: Lots of this stuff is now moved to earlier chapters. This chapter needs to be reworked as a result. It's not clear to me that "chapter3-2.tex" needs to be separate from this.] [RW: New intro blurb needed. Basically, you need to point out how Jigsaw does not suffice, which hopefully you will have demonstrated in the previous chapter. It's not clear to me why this is separate from the next chapter. Perhaps combine them?] [NZ: I COMBINED THEM.]

[NZ: IN THE PREVIOUS SECTIONS I EXPLAINED WHY JIGSAW DOES NOT SUFFICIENTLY ADDRESS OUR PROBLEM.]

In Chapter **??**, 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 structures. I also described how Jigsaw applies this framework on ASTs of a pair of Java methods to determine potential structural correspondences between them in Chapter 4. We now consider how these frameworks could help us (1) to construct an approximation of the best anti-unifier to our problem with special attention to logging calls and (2) to develop a similarity measure between the two ASTs, which can provide us with useful information for anti-unifying a set of ASTs in a later phase. The constructed anti-unifier can be viewed as a structural generalization that represents the commonalities and differences between the two ASTs.

To this end, first we should create an extended form of AST, called AUAST (Anti-unifier AST) that allows the insertion of variables in place of any node in the tree, which is a requirement for HOAUMT (see Section 4.1.1). To approximate the best anti-unifier for our problem, we should

develop a greedy selection algorithm that determines the best correspondence for each node. Our approach contains a sequence of 3 actions to find the best correspondences between two AUASTs, outlined by the algorithm DETERMINE-BEST-CORRESPONDENCES: (1) generates all possible candidate correspondence connections between the two AUASTs using the Jigsaw framework (line 1) (see Section **??**); (2) applies some constraints to prevent the anti-unification of logging calls with anything else (line 2) (see Section 5.1.1); and (3) determines the best correspondence for each AUAST node with the highest similarity and then removes the other correspondence connections involving those nodes (line 3) (see Section 5.1.2). To construct an anti-unifier, a further step should be taken, which is the anti-unification of each AUAST node with its best correspondence through anti-unifying their structural properties (see Section 5.1.3). Furthermore, we computed the ratio of the number of identical simple property values over the total number of simple property values of the anti-unifier to measure similarity between two given AUASTs (see Section 5.1.4).

Algorithm 5.1 DETERMINE-BEST-CORRESPONDENCES($auastA$,$auastB$) determines best correspondences between the two AUASTs $auastA$ and $auastB$

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

To evaluate our anti-unification approach, we have developed the anti-unifier-building tool atop Jigsaw, and conducted an empirical study on a test suite. In Section 5.2, we describe our experimental setup, present our study, and discuss the results.

5.1 The Anti-unifier building tool

The anti-unifier building tool is a proof-of-concept implementation of our anti-unification approach (shown in Figure 5.1), which will be described in the following sections.

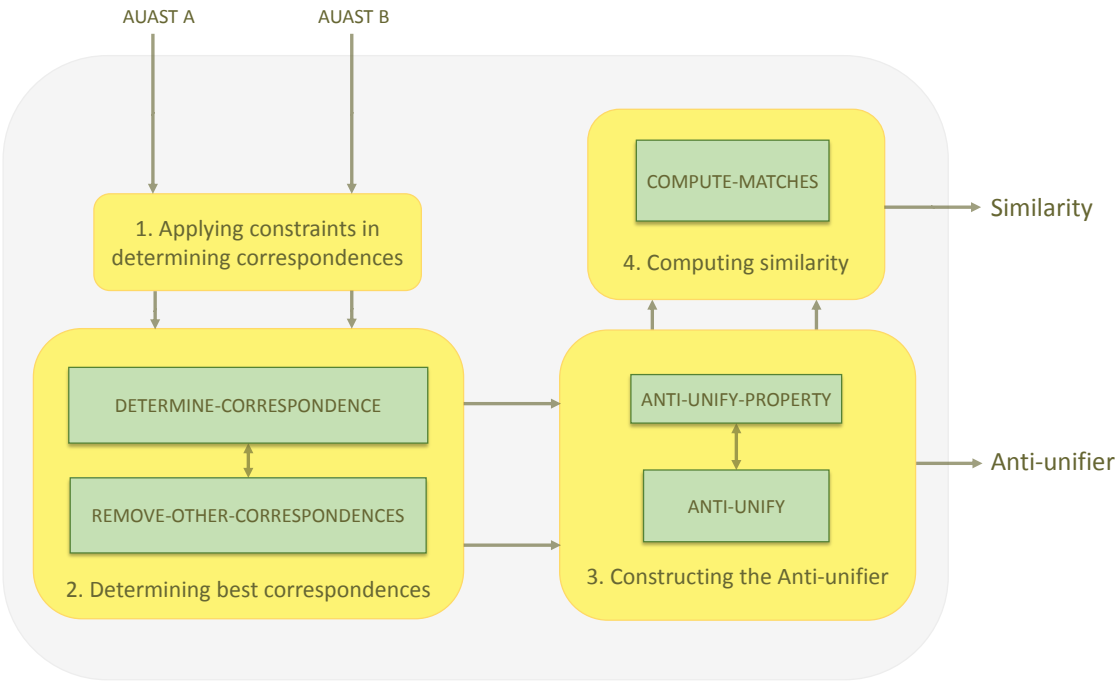


Figure 5.1: Overview of the anti-unification process.

5.1.1 Applying constraints in determining correspondences

To construct an anti-unifier of two AUASTs with a focus on logging calls, some constraints should be applied prior to determining the best correspondences. The first constraint (as described below) should be applied to prevent the anti-unification of log method invocation nodes with any other type of node.

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 AUASTs of two Java classes in Figure 3.14. Jigsaw creates a correspondence connection between the two log method invocation nodes and the two **if** statements. As is clear, the second **if** statement contains a logging call, while there is no corresponding logging call in the first one. According to the first constraint, two log method invocation nodes should be 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.

Our 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 method invocation nodes and structures enclosing them as "logged".

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

5.1.2 Determining best correspondences

As described in Section 3.5, through the application of the Jigsaw framework, each AUAST node will hold a list of candidate correspondence connections, each implicitly representing an anti-unifier. However, despite having multiple potential anti-unifiers, we need to determine one single anti-unifier that is helpful to solve our problem. To do so, we have developed DETERMINE-CORRESPONDENCE 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"in a later step. This algorithm should be used after

applying the constraints mentioned in Section 5.1.1 on candidate correspondence connections.

The DETERMINE-CORRESPONDENCE 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 Jigsaw similarity measure (lines 1–8). The correspondence connection with the highest similarity value is determined as the best fit for the two nodes involved (lines 9–11); all other correspondence connections involving these two nodes are removed using REMOVE-OTHER-CORRESPONDENCES algorithm (line 10). This process terminates when no more correspondence connections are left in the list.

Algorithm 5.2 DETERMINE-CORRESPONDENCE($auastA$) takes in an AUAST and determines the best correspondence connection with the highest similarity for each AUAST node.

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

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 correspondences between AUAST nodes after applying the constraints and the DETERMINE-CORRESPONDENCE algorithm on the list of potential correspondence connections created by the Jigsaw framework in Figure 3.14.

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

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

1: $list1 \leftarrow correspondences[nodeA[ce]]$
2: $list2 \leftarrow correspondences[nodeB[ce]]$
3: **for** $ce1 \in list1$ **do**
4: **if** $ce1 \neq ce$ **then**
5: REMOVE($ce1$, $list$)
6: REMOVE($ce1$, $correspondences[nodeA[ce1]]$)
7: REMOVE($ce1$, $correspondences[nodeB[ce1]]$)
8: **end if**
9: **end for**
10: **for** $ce2 \in list2$ **do**
11: **if** $ce2 \neq ce$ **then**
12: REMOVE($ce2$, $list$)
13: REMOVE($ce2$, $correspondences[nodeA[ce2]]$)
14: REMOVE($ce2$, $correspondences[nodeB[ce2]]$)
15: **end if**
16: **end for**

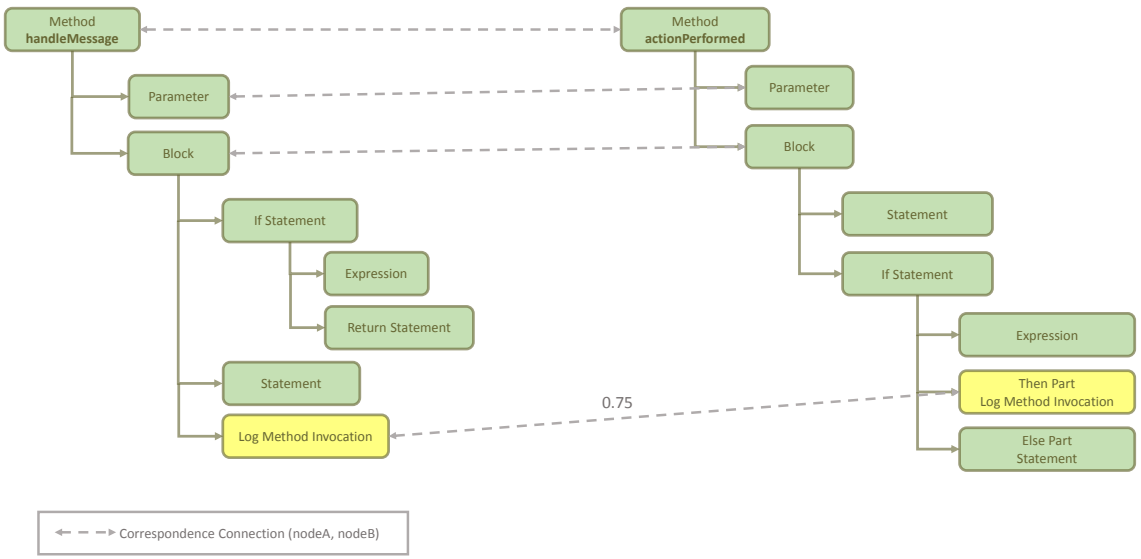


Figure 5.2: Simple AUAST structure of the examples in Figures 3.11 and 3.12. The links between AUAST nodes indicate structural correspondences selected as the best fit by the DETERMINE-CORRESPONDENCE algorithm.

5.1.3 Constructing the anti-unifier

Once the best correspondences have been determined between AUAST nodes, we construct a new anti-unified AUAST by traversing AUAST structures recursively and anti-unifying the structural properties. 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 through the anti-unification of their structural properties, via the ANTIUNIFY algorithm. For each structural property of $auastA$ and $auastB$, where there is no corresponding property in the other AUAST, a structural variable property is created by anti-unifying the structural property with the NIL structure via the ANTIUNIFY-PROPERTY algorithm and is added to properties of the anti-unifier (Lines 4-5 and Lines 13-15); if both nodes have the same property but with different property values, a structural variable property is created via the ANTIUNIFY-PROPERTY algorithm and appended to the anti-unifier structural properties (Lines 6-7); otherwise, if the two nodes have the same exact structural property, a copy of one of them is added to the anti-unifier structural properties (Lines 8-9).

Anti-unification of structural properties $propA$ and $propB$ is performed via the ANTIUNIFY-PROPERTY algorithm. If $propA$ is a simple property, a simple variable property is constructed referring to two simple values (Lines 2-3); if structural property is a child property, a child variable structure is constructed (Line 5); if structural property is a child list property, for each child of $propA$ and $propB$, where there is no correspondence in the other AUAST, an anti-unified node is created through anti-unifying the child node with the NIL structure via the ANTIUNIFY algorithm and added to the value of the anti-unified child list property; otherwise, the child node is anti-unified with its best correspondence (Lines 6-20).

For example, we supply the ANTIUNIFY algorithm with the AUASTs of LJMs in Examples 1

Algorithm 5.4 Inputs into ANTIUNIFY($auastA$, $auastB$) are two AUASTs. This algorithm construct an anti-unified AUAST through the anti-unification of input nodes' structural properties.

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

and 2. Figure 5.3 shows a simple detailed view we used to represent the anti-unifier constructed from the two AUASTs, where "a-or-b" represents a structural variable that must be substituted with either a or b substructures to recover each original structure, and "a" represents that a substructure is common between the two AUASTs.

Algorithm 5.5 ANTIUNIFY-PROPERTY($propA$, $propB$) takes two structural properties and constructs an anti-unified structural property.

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

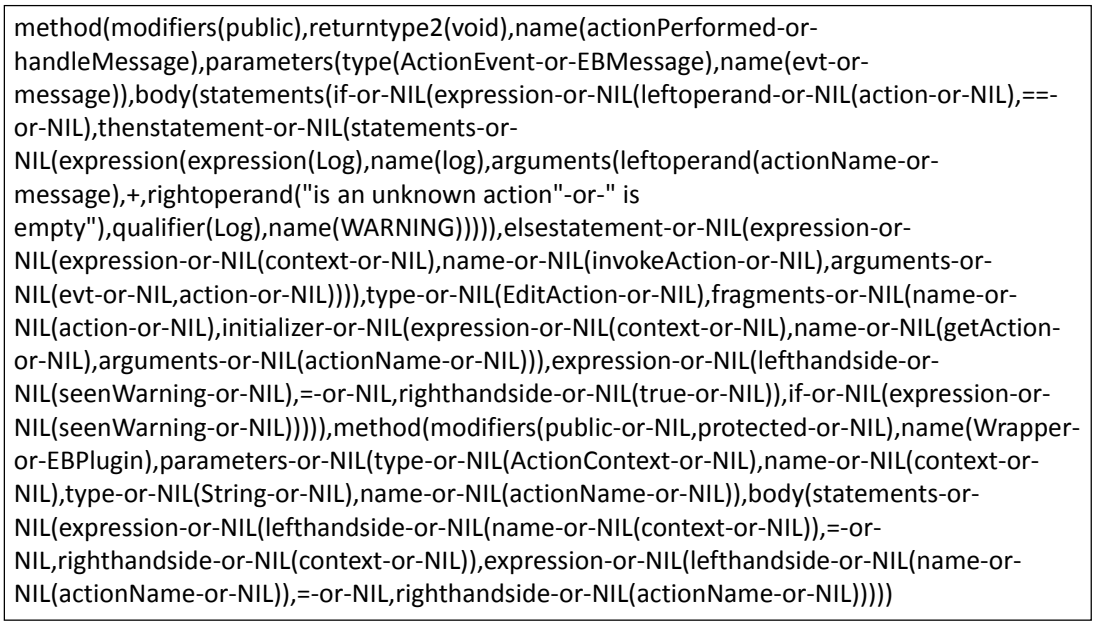


Figure 5.3: Detailed view of the anti-unifier constructed from the AUASTs of the LJMs in Examples 1 and 2.

5.1.4 Computing similarity between AUASTs

Similarity computation is particularly important for clustering phase that relies on accurate estimation of similarity between logged Java methods and will be discussed in detail in Section 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 Java methods that use logging calls based on the structural similarity between them. The structural similarity of two AUASTs can be defined as the number of identical simple structural property values over the total number of simple structural property values of the anti-unifier.

Algorithm 5.6 COMPUTE-MATCHES($auastA$) takes in one of the AUASTs and determines the matches between the two AUASTs via a recursive traversal of structural properties.

```
    COMPUTE-MATCHES(auastA)
 1: matches ← 0
 2: for property ∈ properties[auastA] do
 3:     valueA ← value[property]
 4:     valueB ← GET-BEST-CORRESPONDENCE(valueA)
 5:     if property instanceof SimpleProperty then
 6:         matches ← matches + JIGSAW-MATCHES(valueA, valueB)
 7:     else if property instanceof ChildProperty then
 8:         matches ← matches + COMPUTE-MATCHES(valueA)
 9:     else if property instanceof ChildListProperty then
10:         for nodeA ∈ valueA do
11:             nodeB ← GET-BEST-CORRESPONDENCE(nodeA)
12:             matches ← matches + COMPUTE-MATCHES(nodeA)
13:         end for
14:     end if
15: end for
16: return matches
```

The number of identical simple structural property values between $auastA$ and $auastB$ is computed via the COMPUTE-MATCHES algorithm through a recursive traversal of $auastA$ nodes' structural properties. For simple structural properties, the number of matches is computed re-using the Jigsaw similarity function that computes the number of matches between the property values (Lines 5-6). For child structural properties, the number of matches is computed recursively for a child node and is propagated to the parent(Lines 7-8). For child list structural properties, the number of matches is computed for each child node recursively and is propagated to the parent node (Lines 9-14). All matches are summed up to compute total number of matches between the two AUASTs. Then the following equation is used to compute structural similarity between $auastA$ and $auastB$:

$$similarity = \frac{2 * matches}{|auastA| + |auastB|} \tag{5.1}$$

The similarity function returns a value between 0 and 1 that indicates zero and total matching, respectively.

5.1.5 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 a logged Java method. For example, consider the logged Java methods in Figures 5.4 and 5.5. Figure 5.6 shows the simple AUASTs for these examples and potential correspondence connections between 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 "nothing" 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).

```
1  public void method1(){
2          ...
3          if (condition1){
4                  Log.log();
5          }
6          ...
7          if (condition2){
8                  Log.log();
9          }
10         ...
11 }
```

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

```
1  public void method2(){
2          ...
3          if (condition3){
4                  Log.log();
5                  Log.log();
6          }
7          ...
8  }
```

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

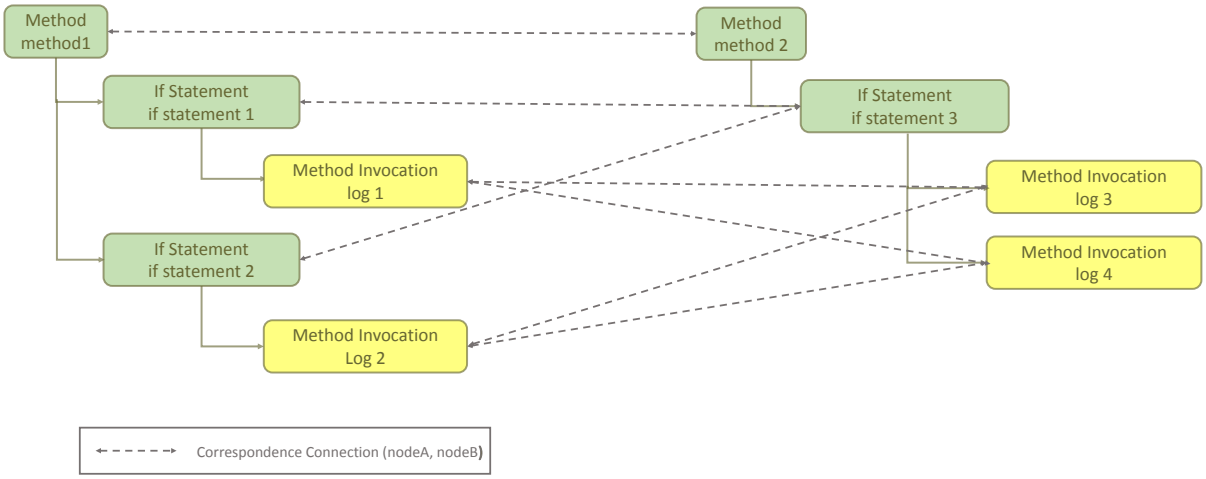


Figure 5.6: Simple AUAST structure of the examples in Figures 5.4 and 5.5. Links between AUAST nodes indicate candidate structural correspondences detected by the Jigsaw framework.

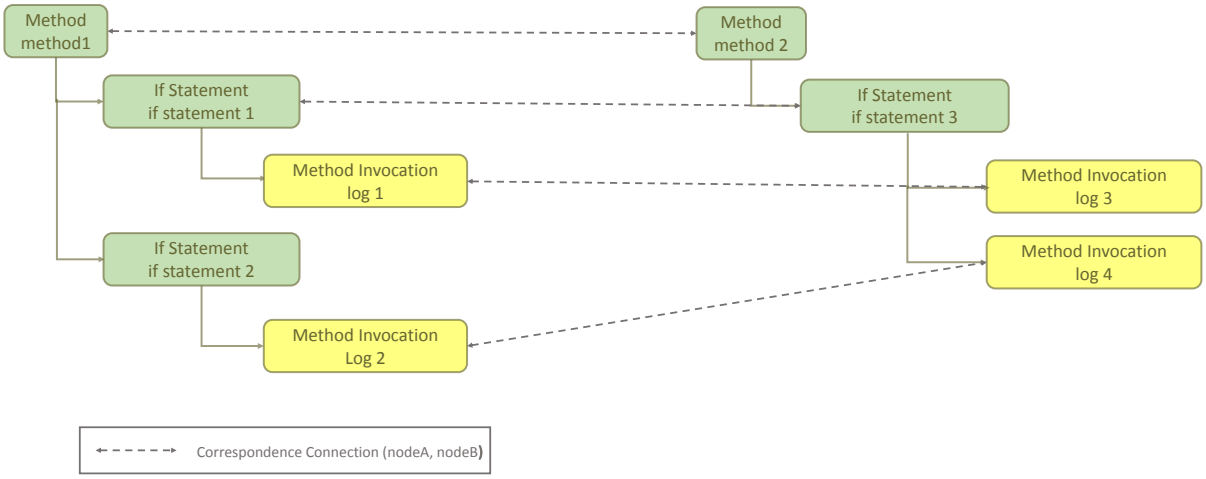


Figure 5.7: Simple AUAST structure of the examples in Figures 5.4 and 5.5. Links between AUAST nodes indicate structural correspondences selected as the best match using our greedy algorithm.

To handle these cases, we can split them into more than one case, where each logged Java method contains only one logging call. To do so, we need to create a copy of logged Java method 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.

```
1
2 public void method1(){
3             ...
4             if (condition1){
5                         Log.log();
6             }
7             ...
8             if (condition2){
9                         //removed
10            }
11            ...
12 }
13
14 public void method1(){
15            ...
16            if (condition1){
17                        //removed
18            }
19            ...
20            if (condition2){
21                        Log.log();
22            }
23            ...
24 }
```

Figure 5.8: Create multiple copies of the LJM in Figure 5.4 for each logging call.

```
1 public void method2(){
2         ...
3         if (condition3){
4                 //removed
5                 Log.log();
6         }
7         ...
8 }
9
10 public void method2(){
11        ...
12        if (condition3){
13                Log.log();
14                //removed
15        }
16        ...
17 }
```

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

5.2 An assessment of the anti-unifier-building tool

To assess the effectiveness of our anti-unification algorithm, we have implemented the anti-unifier-building tool, which is a plug-in to the Eclipse integrated development environment (IDE), and conducted an experiment on the test suite described in Section 4.2. Our tool is developed atop Jigsaw to construct an anti-unifier for each pair of LJMs in our test suite.

5.2.1 Setup

In this study, we manually attempted to create the detailed anti-unifier view for each pair of LJMs in the test suite (55 test cases in total). We first identified corresponding and non-corresponding Java elements for each LJM 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 LJMs to measure the similarity. We also ran the anti-unifier-building tool on each pair of LJM to construct the detailed anti-unifier view for each

pair with special attention to logging calls and to measure the similarity between the two LJMs. Furthermore, we used Eclemma, which is a Java code coverage tool for Eclipse, to measure the test coverage. Test coverage is defined as a measure of the completeness of the set of test cases.

5.2.2 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 of LOC of the two LJMs. ”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 Test Case 8, rootEntry method contains a nested **if** – statement enclosing a logging call and 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. Test Cases 3 and 10 contain statements that are not found to be corresponded by our tool even though correspondences exist. For example, in Test Case 3, isSupportedEncoding method contains an assignment statement enclosed by an **if** – statement that does not have any correspondences and 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 are not corresponded.

The results of the pairwise comparison between LJMs 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 test coverage of our test cases was measured 82% using EclEmma.

Test case	Logged Java methods
1	org.gjt.sp.jedit.PluginJAR.generateCache()
	org.gjt.sp.jedit.PluginJAR.generateCache()
2	org.gjt.sp.jedit.PluginJAR.generateCache()
	org.gjt.sp.jedit.EditBus.send(...)*
3	oorg.gjt.sp.jedit.MiscUtilities.isSupportedEncoding(...)
	org.gjt.sp.jedit.EditBus.send(...)
4	org.gjt.sp.jedit.EditBus.send(...)
	org.gjt.sp.jedit.EditBus.send(...)*
5	org.gjt.sp.jedit.EditBus.send(...)*
	org.gjt.sp.jedit.EditAction.Wrapper.actionPerformed(...)
6	org.gjt.sp.jedit.EditBus.send(...)*
	org.gjt.sp.jedit.BufferHistory.RecentHandler.doctypeDecl(...)
7	org.gjt.sp.jedit.EditAction.Wrapper.actionPerformed(...)
	org.gjt.sp.jedit.JARClassLoader.loadClass(...)
8	org.gjt.sp.jedit.EditAction.Wrapper.actionPerformed(...)
	org.gjt.sp.jedit.io.VFS.DirectoryEntry.RootsEntry.rootEntry(...)
9	org.gjt.sp.jedit.PluginJAR.generateCache()
	org.gjt.sp.jedit.BufferHistory.RecentHandler.doctypeDecl(...)
10	org.gjt.sp.jedit.io.VFS.DirectoryEntry.RootsEntry.rootEntry(...)
	org.gjt.sp.jedit.ServiceManager.loadServices(...)

Figure 5.10: 10 sample logged Java method pairs used as test cases.

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.

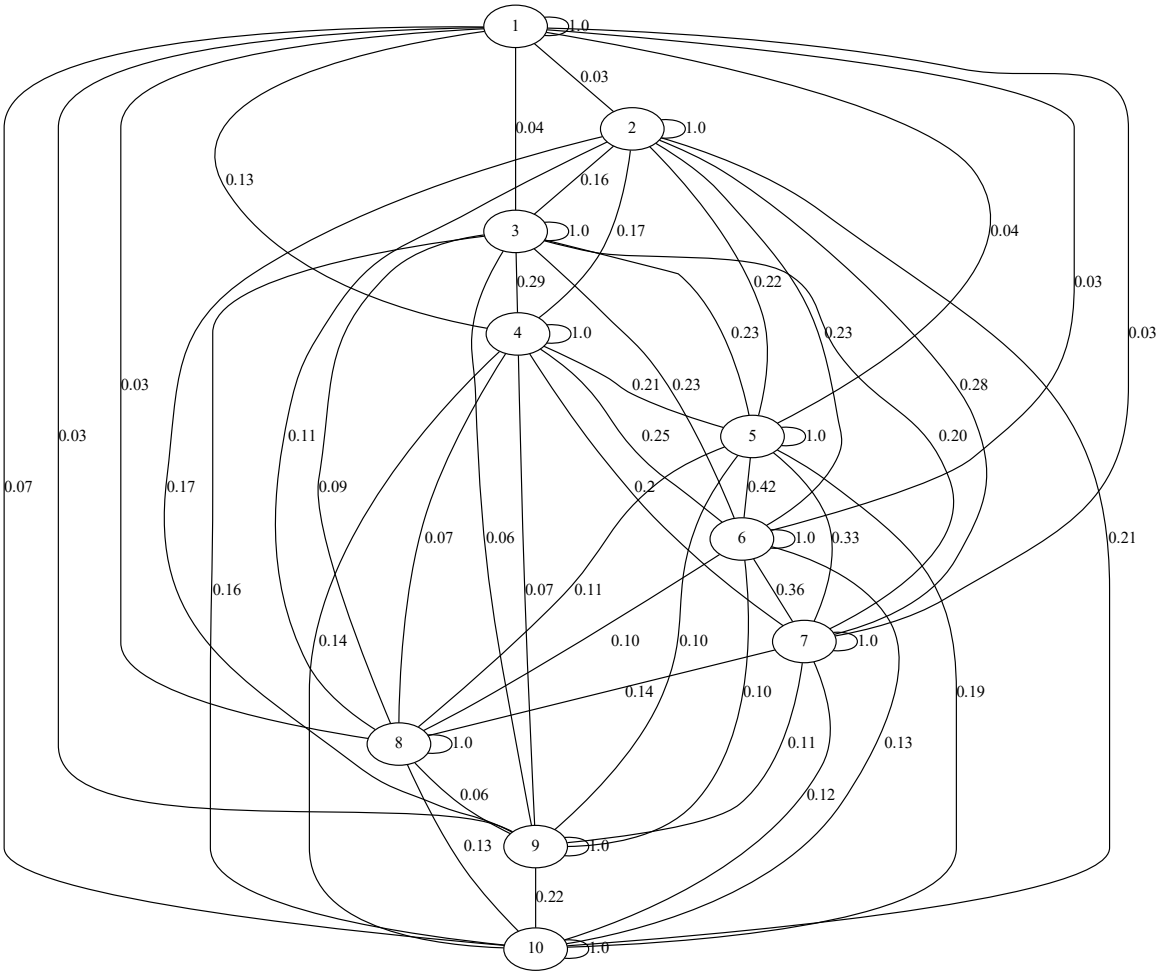


Figure 5.12: A similarity graph representing pairwise similarities calculated by our tool between LJMs shown in Table 4.2.

5.3 Summary

We have presented an approach for constructing a generalization from ASTs of two logged Java methods with special attention to logging calls. This approach is implemented as an Eclipse plug-in which given two logged Java methods utilizes the Eclipse JDT framework to extract their ASTs. In order to be able to apply HOAUMT, we extended the AST structure to a higher-order structure, called AUAST, that would allow the insertion of variables in place of any nodes. We then applied

the Jigsaw framework to identify potential correspondences between the two AUASTs and greedily determines the best correspondence for each node with the highest Jigsaw similarity. Moreover, some constraints have been applied on the selection of correspondences to prevent anti-unifying log method invocation nodes with any other types of node. The anti-unification of two AUASTs is performed through the application of higher-order modulo theories over the AUAST structures. A measure of similarity has been developed that would provide us with useful information for the clustering phase. Furthermore, an empirical study was conducted to evaluate the effectiveness of our anti-unification algorithm and the anti-unifier-building tool in constructing an anti-unifier from each LJM pair of our test suite with special attention to logging calls and measuring similarity between them.

Chapter 6

Clustering

[NZ: I WOULD BE GRATEFUL IF YOU COULD HELP ME TO CHOOSE AN APPROPRIATE TITLE FOR THIS CHAPTER. IT DESCRIBES MY APPROACH TO CLASSIFY LJMS, ITS IMPLEMENTATION, AND THE EXPERIMENT I CONDUCTED TO EVALUATE IT.]

In Chapter 5, we described our anti-unification algorithm to construct an anti-unifier from AUASTs of a pair of LJMs with a 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 the source code by constructing structural generalizations that represent the detailed structural similarities and differences of LJMs. To this end, we should develop an algorithm that:

- classifies AUASTs of LJMs into groups using a measure of similarity such that AUASTs in each group has maximum similarity with each other and minimum similarity to other ones.
- abstracts AUASTs of each group into a structural generalization representing the similarities and differences between them.

To construct an anti-unifier and to develop a measure of similarity, we can utilize the anti-unification algorithm presented in Chapter 5. However, to produce structural generalizations from AUASTs of a set of LJMs a further should be taken, which is applying a clustering algorithm that would allow the classification of AUASTs into groups. To this end, we have developed a modified version of an agglomerative hierarchical clustering algorithm.

Our hierarchical clustering algorithm (Section **??**) is a bottom-up approach that starts with singleton clusters, where each contains one AUAST. In every iteration, it merges the closest clusters which are the clusters with maximum similarity between their AUASTs. We use the similarity function described in Section 5.1.4 to measure similarity between each pair of AUASTs and then

construct an anti-unifier via the anti-unification algorithm described in Section 5.1.3 when it is needed to merge two clusters.

To evaluate our clustering approach, we have developed a tool atop the anti-unifier-building tool, and conducted an empirical study on our test suite. In Section **??**, we will describe our experimental study and discuss the results.

6.1 The Clustering tool

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

1. Start with singleton clusters, where each cluster contains one AUAST
2. Compute the similarity between clusters in a pairwise manner
3. Adjust the similarity between a cluster pair to zero, if the anti-unification of their AUASTs does not allow anti-unifying log method invocation nodes with each other as the structures enclosing them are not corresponded.
4. Find the closest clusters (a pair of clusters with maximum similarity)
5. Merge the closest cluster pair and replace them with a new cluster containing the anti-unifier of AUASTs of the two clusters
6. Compute the similarity between the new cluster and all remaining clusters

- Repeat Steps 3, 4, 5, and 6 until the similarity between closest clusters becomes below a predetermined threshold value

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. The step 3 is inserted to prevent the combination of clusters when the usage of logging in their AUASTs has not been

performed in the same way and they should be in separate clusters. Furthermore, the similarity threshold is determined through informal experimentation.

Figure 6.1 illustrates the clustering process for classifying a set of 4 AUASTs. In the first iteration, the closest clusters, which are cluster 1 and cluster 2, are merged and replaced by cluster 5. If threshold value is determined as threshold A, the process will be terminated as the similarity between the closest clusters is below this threshold;if not, cluster 3 and cluster 5 will be merged and replaced by cluster 6. However, the similarity between AUASTs of cluster 6 and 7 is zero, thus they should not be merged with each other.

The $n \times n$ similarity matrix is shown in Figure **??** where an element in row i and column j represents the similarity between the i^{th} and the j^{th} clusters .

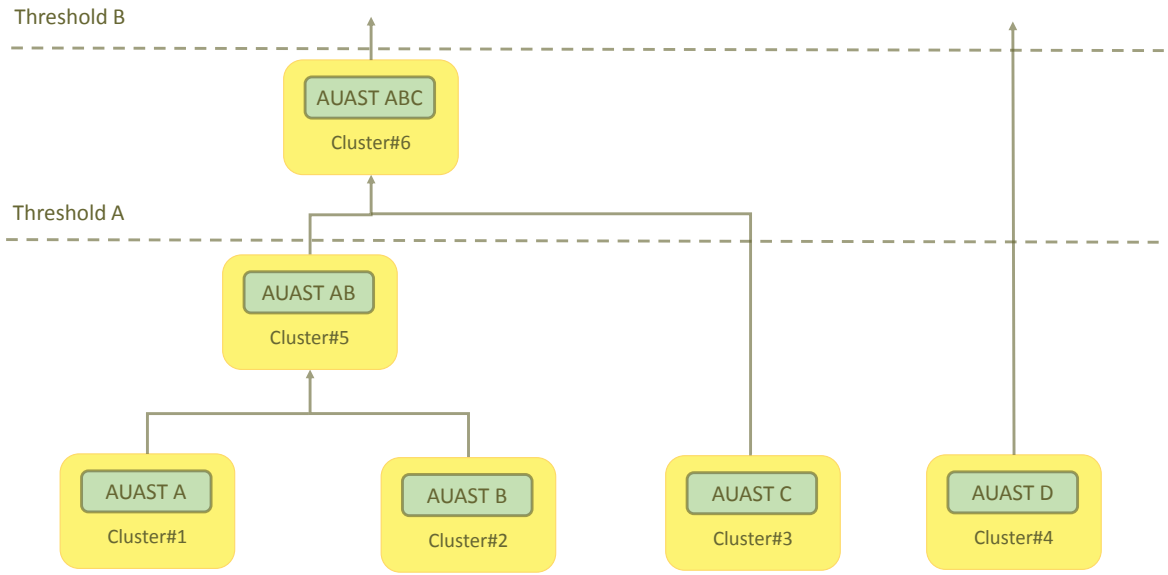


Figure 6.1: The agglomerative hierarchical clustering process for classifying 4 AUASTs. The threshold value indicates the number of clusters we will obtain.

Cluster	Correspondence		Similarity		Relevancy	
	Correct (%)	Incorrect	human	tool	human	tool
1	28(100)	0	0.09	0.09	4	4
3	24(92)	2	0.19	0.2	3	3
2	9(100)	0	0.25	0.25	3	3

Figure 6.2: Results from applying the clustering tool to the test suite.

6.2 An Assessment of the Clustering tool

To assess the effectiveness of our clustering algorithm in classifying a set of AUASTs, we have implemented the clustering tool, which is a plug-in to the Eclipse integrated development environment (IDE), and conducted an experiment on the set of AUASTs of LJMs in the test suite described in Section 4.2. The tool is developed atop the anti-unifier-building tool.

6.2.1 Setup

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

6.2.2 Results

We present the results of our analysis in Table 6.2. The analysis of the output has been divided into three categories: correspondence, similarity, and relevancy. The analysis of correspondence and similarity was described in Section 5.2.2. ”Relevancy” represents that the usage of logging in Java methods of each cluster is similar that they should be grouped in the same cluster, and dissimilar to the other Java methods in the other clusters that should be scattered in different clusters.

Our clustering tool succeeded in detecting the relevancy between the LJMs of our test suite. It also successfully calculated the similarity between LJMs 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

We have presented a modified version of the agglomerative hierarchical clustering algorithm to classify AUASTs of a set of LJMs using a measure of similarity. This algorithm is implemented as an Eclipse plug-in, which given a set of LJMs utilizes the anti-unifier building tool to measure similarities between AUASTs of each cluster pair and to construct an anti-unifier from them when two clusters need to be merged. Furthermore, an empirical study was conducted to evaluate the effectiveness of our clustering algorithm and its implementation in classifying AUASTs of the set of LJMs in our test suite.

Chapter 7

Evaluation

- Two empirical studies were conducted

7.1 Experiment 1

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

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

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

7.2 Experiment 2

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

7.3 Results

- I will describe the results taken form the second experiment

7.4 Lessons learned

- I will describe our findings

Chapter 8

Discussion

In this chapter, we discuss the validity of our evaluation and the characterization study (Section 8.1), the limitations and pitfalls of the approach and our tool support (Section 8.2), and a number of remaining issues including: the other applications of our tool's underlying framework (Section 8.3); the usage of anti-unification theory (Section 8.4).

8.1 Threats to validity

Prior to applying our tool for characterizing logging usage in real-world software systems, we have conducted three experiments to investigate the effectiveness of our proposed approach. However, there are several potential threats regarding the validity of these experiments. First, the results of our manual examination might be biased, as I determined the correct correspondences between ASTs and the correct way of classifying the set of ASTs 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 LJMs from a real-world software system, though different test suites may generate different results. Although I cannot claim that the LJMs in my test suite are a good representative of all LJMs 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 happen accidentally only for our test suite. To resolve this doubt, we examined the cases where our tool fails to detect correct correspondences and we 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, or software written in other programming languages.

8.2 Our tool output

In addition, there are some issues that the approximation approach and our tool support is not able to handle perfectly, including node ordering mismatch, and the management of conflicts happened in constructing the anti-unifiers.

8.2.1 Node ordering mismatch

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 Handling conflicts in constructing the anti-unifiers

The decisions we have made to resolve the conflicts occurred in constructing structural generalizations might affect the accuracy of our results. For example, in situations where we have two correspondences with the same similarity value in the ordered list of correspondence connections, our 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, we consider AST hierarchies to perform anti-unification. That is, our 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 leads us to get approximate results, they helped to limit the complexity of our approach allowing the implementation of it as a practical solution.

8.3 Other applications

Any applications that are involved in the inference of structural patterns in source code even infrequently-used patterns might benefit from our tool's underlying framework. Furthermore, understanding the commonalities and differences between source code fragments has application in several areas of software engineering, such as code clone detection, API usage pattern collation, 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 to represent the similarities and differences between a set of source code fragments via structural correspondence could be used to improve the results of these studies as well.

8.4 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, which is useful for 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.

The problem of my study is different, but this work shows how the Jigsaw framework could be useful in constructing structural generalizations to solve a particular problem context via the determination of structural correspondences between two ASTs. The application of HOAUMT to construct structural generalizations via structural correspondences is novel to the problem of extracting usage patterns of log statements in source code. Finally, we made a comparison between our tool and the Jigsaw tool, which were developed for various applications, but part of the implementation of Jigsaw was successfully applied to solve our problem context.

8.5 Summary

We discussed the potential threads to validity of our evaluation and characterization study. To limit the bias of the experiments we conducted to evaluate the effectiveness of our approach and the tool support, we selected our test cases form a real system with various levels of similarity in the usage of logging calls. Furthermore, we examined the failed test cases to assure that our tool works when

it should work with regard to the proposed algorithm. We will also make our test suite available for public examination to check the accuracy of our manual inspection. For our characterization study, we selected various software systems in terms of functionality that are widely used by many developers for a long period of time. We also discussed the remaining issues with our tool support including node ordering, , and that can be resolved in future as the extensions of our work.

This work aim to provide a detailed view of structural generalizations constructed from a set of source code fragments that use log statements via structural correspondence and clustering. However, we discussed that any other applications involved in the inference of structural usage patterns of a particular statement or the detection of commonalities and differences between a set of source code fragments could benefit from our tool's underlying framework.

We also 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 the source code (Section 9.2), data mining approaches to extract API usage patterns (Section 9.3), anti-unification and its application to detect strcutural 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 the 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 behavior understanding:** Fu et al. [2013] present an approach for understanding system behavior through contextual analysis of logs. They first extracted execution patterns reflected by a sequence of system logs and then utilized the patterns to find contextual factors from logs that causes a specific system behavior. The Linux Trace Toolkit [Yaghmour and Dagenais, 2000] was created to record and analyze system behavior by providing an efficient kernel-level event logging infrastructure. A

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

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

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

Jiang et al. [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 a great effort to modify logging calls as after-thoughts, which indicates that they are not satisfied with the log quality in their first attempt. They also characterize where developers spend most of their time in modifying the log messages.

Yuan et al. [2012a] studies 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 message by detecting important variable values and inserting them into the log messages. However, these studies only consider code snippets containing bugs that are needed to be logged and do not consider other code snippets containing no bugs but still need to be logged. Moreover, these studies mainly research log message modifications and potential enhancements of

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

9.2 Correspondence

Several studies have been conducted to find similarities and differences between the source code fragments. Baxter et al. [1998] develop an algorithm to detect code clones in source code that uses hash functions to partition subtrees of ASTs of a program source code and then find common subtrees in the same partition through a tree comparison algorithm. Apiwattanapong et al. [2004] present a top-down approach to detect differences and correspondences between two versions of a Java program, through comparison of the control flow graphs created from the source code. Holmes et al. [2005] recommends relevant code snippet examples from a source code repository for the sake of helping developers to find examples of how to use an API by heuristically matching the structure of the code under development with the source code in the repository. Coogle [Sager et al., 2006] is developed to detect similar Java classes through converting ASTs to a normalized format and then comparing them through tree similarity algorithms. However, none of these approaches determines the detailed structural correspondences needed in our context.

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

9.3 API usages patterns

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

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

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

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, k-means clustering algorithm is not a good fit to our problem since it requires to predefine the number of clusters we want to come up with, which is not reasonable in our context.

Hierarchical clustering algorithms produce a nested grouping of clusters, with single point clusters at the bottom and an all-inclusive cluster at the top [Karypis 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 is below a pre-determined threshold value. Hierarchical clustering algorithms work implicitly or explicitly with the $n \times n$ similarity matrix such that an element in row i and column j represents the similarity between the i^{th} and the j^{th} clusters [Karypis 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]. In the single linkage method, the similarity is measured by the similarity of the closest pair of data points of the two clusters. In the complete linkage method, the similarity is computed by the similarity of the farthest pair of data points of the two clusters. In the average linkage method, the similarity is measured by the average similarity of all pairwise similarities of data points of the two clusters. In the centroids methods, each cluster is represented by a centroid of all data points in the cluster, and the similarity between two clusters is measured by the similarity of the clusters' centroids. However, in our application, each cluster is composed of one AUAST, and the similarity between two clusters is measured by the similarity between the clusters' AUASTs, which is computed via anti-unification.

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

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

Chapter 10

Conclusion

Determining the detailed structural similarities and differences between a set of source code fragments is a complex task, and it 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. We have presented an approach to characterize where logging calls happen in source code by means of structural generalization and clustering. 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 and determines potential structural correspondences between the AST nodes via the Jigsaw framework. Second, it constructs an anti-unifier form ASTs of two given LJMs 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.

We have conducted three experiments to evaluate the effectiveness of our approach in constructing structural generalizations and classifying Java methods that use logging calls. I found that my tool was successful in determining correct correspondences for my application in % of

test cases. It was also successful in classifying logged Java methods into separate clusters using a similarity measure that indicates how similar logged Java methods are with a focus on the usage of logging calls. Furthermore, An study was conducted to describe the commonalities and differences between the usage of logging calls in the source code of three software systems via our tool that describes logging usage patterns on a per system and between systems method-granularity basis. Our characterization study shows

In summary, our study makes the following contributions:

- An approach to construct a structural generalization from AST structures of two logged Java methods with special attention to logging usage by determining structural correspondences between the ASTs via the Jigsaw framework and an approximated higher-order anti-unification modulo theories algorithm.
- An approach to develop a similarity measure that determines how similar two logged Java methods are with a focus on logging usage.
- An approach for classifying a set of ASTs via a hierarchical clustering algorithm.
- An approach for detecting usage patterns of logging calls in source code via structural generalization and clustering.

10.1 Future Work

Future extensions could be applied to resolve the remaining problems of this study:

- Data flow analysis techniques: to resolve the problem of inaccurate node ordering.
- Further analysis: to detect and resolve all the conflicts happen in deciding the best correspondences.

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. We believe that 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.

To further validate the findings of our characterization study, the source code analysis can be performed on more software systems. 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.

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.

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 *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 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 22nd international conference on Software engineering*, pages 167–176. ACM, 2000.

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.

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 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.

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.