Characterization of Logging Usage: An Application of Discovering Infrequent Patterns via Anti-unification

- Determining the detailed structural similarities and differences between entities of the source code of a software system, or between the source code of different software systems, is a potentially complex problem
- Being able to do so has various actual or potential applications
- As a specific application, our focus is on the study of where logging is used in the source code
- logging is a pervasive practice and has various applications in software development and maintenance
- Developers have to make decisions about where and what to log
- Logging should be done in an appropriate manner to be effective
- Researches have often considered logging as a trivial task
- However, some evidence suggest that it is not a straightforward task to perform high quality logging in practice, such as:
  - availability of several complex frameworks to help developers to log
  - the significant amount of effort developers spend to modify logging calls as after-thoughts [Yuan et al., 2012]
- So far, little study has been conducted on understanding the usage of logging in real-world systems
- PROBLEM: In this research, we would like to understand where developers log in practice, in a detailed way
- SOLUTION: Develop an automated approach to detect the detailed structural similarities and differences in the usage of logging within a system and between systems
- [NZ: I am a little confused about how to find commonalities and differences of logging usage between systems? do you mean that I should compare the results taken from clustering of all Java classes in one system to another one?] **[RW: The process of locating commonalities and differences can be applied within a single version of one system, across multiple versions of one system, across single versions of multiple systems, or across multiple versions of multiple systems. It would be useful to understand the differences and similarities between different systems. Is there some reason that this would be harder to achieve?]** [NZ: I should think about it more. I can answer to this question after per-system analysis with more confidence]
- **Section 1.1 Broad thesis overview**

- We aim to provide a concise description of where logging calls are used in the source code through creating generalizations that represent the detailed structural similarities and differences between Logged Java Classes (LJCs)
- Our approach:
  - applies a hierarchical clustering algorithm to classify LJCs into groups using a measure of similarity
  - uses an anti-unification algorithm to construct a structural generalization representing the similarities and differences of all LJCs in each group
- Our anti-unification approach:
  - uses the Jigsaw framework to determine all potential correspondences between a pair of LJCs
  - applies some constraints to avoid anti-unifying logged Java classes with non-logged Java classes
  - determines correspondences between structures containing logging calls by greedily applying a similarity measure to find the most similar substructures
  - develops a measure of structural similarity between LJCs
- Our approach has been implemented as an Eclipse plug-in
- An experiment is conducted on 10 sample LJCs to evaluate our approach and tool
- To characterize logging usage, we applied our tool on the source code of three open-source software systems that make use of logging
- Our experiment shows ...
- **Section 1.2 Overview of related work**
- Yuan et al. [2012] provides a quantitative characteristic study of log messages
- So far, anti-unification has been used for various applications:
  - generalization task [Cottrell et al., 2007]
  - semi-automating small-scale reuse task[Cottrell et al., 2008]
  - Software clone detection [Bulychev and Minea, 2008]
- This study makes the first attempt to characterize where logging calls occur through finding the detailed structural similarities and differences using anti-unification
- **Section 1.3 Thesis statement**

- The thesis of this work is to determine the detailed structural similarities and differences between entities of the source code that make use of logging to provide a concise description of where logging do occur in real systems
- **Section 1.4 Thesis Organization**
- Chapter 2: motivates the problem of understanding where to use logging calls in the source code through an example
- Chapter 3: provides background information on ASTs, anti-unification and its extensions (HOAUMT), and the Jigsaw framework and why they matter in our study
- Chapter 4: describes our proposed approach and its implementation as an Eclipse plug-in
- Chapter 5: presents an empirical study conducted to evaluate our approach and its application to characterize logging usage
- Chapter 6: describes related work to our research problem and how it does not adequately address the problem
- Chapter 7: discusses the results and findings of my work, threats to its validity, and remaining issues
- Chapter 8: concludes the dissertation, presents the contributions of this study, and future work
- **Chapter 2. Motivational Scenario**
- Logging is a systematic way of recording the software runtime information
- A typical logging call is composed of a log function and its parameters including a log message and verbosity level
- Consider a developer is given the task of logging the Java class in Figure 1
- She has to make several decisions about
  - what events need to be logged?
  - where to use logging calls?
  - how to decide on the log message and verbosity level of each logging call?
- It is recommended to simply log at the start and end of every method
- However, for example, logging at the start and end of the method *addToBus* is useless, producing redundant information
- She needs more information to perform logging appropriately
- Having a characterization of how usually developers use logging calls in similar situations would assist her in making decisions

- For example, knowing that developers use logging calls inside of if statements to log a potential error when a variable contains an incorrect value, she adds an if statement to log an error when the value of the variable *time* is *null* (shown in Lines 36-38 of Figure 2)
- For example, knowing that developers use logging calls inside catch blocks to record an exception, she creates a try/catch block to capture the potential failure in sending messages and uses a logging call in the catch block (Lines 41-43 of Figure 2)
- Using a concise characterization she would be able to make informed decisions about where to use logging calls more easily and quickly
- With taking appropriate decisions about where to use logging calls, she can spend more time and energy to write the context of log functions
- **Chapter 3. Background**
- **Section 3.1. Abstract Syntax Tree**
- The Eclipse Java Development Tools (JDT) framework provides APIs to access and manipulate Java source code via Abstract Syntax Tree (AST)
- AST maps Java source code in a tree structure form
- Structural properties of an AST node hold specific information for the Java element it represents
- Structural properties are stored in a map data structure that associates each property to its value
- The structural properties have three different types:
  - *Simple Property:* where the value is an AST constant (simple value)
  - *Child Property:* where the value is a single AST node
  - *Child list property:* where the value is a list of AST nodes
- **Section 3.2. Anti-unification**
- A *term* is a set of
  - *Function symbols* (e.g., *f(a,b)*)
  - *Constants* (e.g. *a*)
  - *Variables* (e.g., X)
- *Ground term*: a term that does not contain any variables
- A *Substitution* is defined on a term to map its variables to ground terms
- *Applying Substitutions*: replacing all occurrences of a variable with a proper term

- *Instance & Anti-Instance* : a is an instance of X and X is an anti-instance of a, if there is a substitution $\ominus$ such that the application of $\ominus$ on X results in a ($X \xrightarrow{\ominus} a$)
- *Unifier*: A common instance of two given terms
- *Most General Unifier (MGU)*: U is MGU of two structures such that for all unifiers $U\prime$ there exist a substitution $\ominus$ such that $U \xrightarrow{\ominus} U\prime$
- *Generalization*: X is a generalization for a and b, where X is an anti-instance for a and b under substitutions $\ominus_1$ and $\ominus_2$, respectively. ($X \xrightarrow{\ominus_1} a$ and $Y \xrightarrow{\ominus_2} b$)
- *Anti-unifier*: A common generalization of two given terms
- *Most Specific Anti-unifier (MSA):* A is MSA of two structures where there exist no anti-unifier $A\prime$ such that $A \xrightarrow{\ominus} A\prime$
- MSA should preserve as much of structure of both original structures as possible
- *Anti-unification*: The process of finding the MSA of two given terms
- *First-order anti-unification*: restricts substitutions to replace only first-order variables by terms
- PROBLEM: First-order anti-unification fails to capture complex structural commonalities
- SOLUTION: Higher-order anti-unification allow the creation of MSA by extending the set of possible substitutions such that variables can be replaced by not only constants but also functional symbols
- An instance of an AST structure can be mapped to our recursive definition of a term, where AST nodes and simple values may be viewed as function-symbols and constants, respectively
- A set of equivalence equations is defined to incorporate semantic knowledge of structural equivalences supported by the Java language specification
- These equational theories are applied on higher-order extended structures to allow the anti-unification of AST structures that are not identical but are semantically equivalent
- The NIL structure is defined to create a structure out of nothing that would allow the anti-unification of two structures when a substructure exist in one but not in the other one
- Defining complex substitutions results in losing the uniqueness of MSA
- The complexity of finding an MSA is undecidable in general since an infinite number of substitutions can be applied on every variable in a structure

- Our goal is to find an MSA that is an approximation of the best fit to our application
- **Section 3.3. Jigsaw**
- The Jigsaw tool is developed by Cottrell et al. [2008] to determine the structural correspondences between two Java source code fragments through anti-unification such that one fragment can be integrated to the other one for small scale code reuse
- The Jigsaw framework could help us to construct an anti-unifier form logged Java classes as it:
  - applies higher-order anti-unification modulo theories to address the limitations of first-order syntactical anti-unification
  - creates an augmented form of AST, called CAST (Correspondence AST), in which each node holds a list of candidate correspondence connections between the two structures, each representing an anti-unifier
  - develops a measure of similarity to indicate how similar the nodes involved in one correspondence connection are
- The Jigsaw similarity function:
  - returns a value between 0 and 1 where indicate zero and total matching, respectively
  - uses several semantical heuristics to improve the accuracy of similarity measurement
- The Jigsaw tool does not sufficiently solve our problem since:
  - it determines all potential correspondences between AST nodes, each representing an anti-unifier; however, we need to determine one single anti-unifier
  - it does not construct an anti-unifier of two given AST structures with a focus on logging calls
- **Chapter 4. Anti-unification of Logged Java Classes**
- **Section 4.1. Constructing an anti-unifier**
- Problem: constructing an anti-unifier (structural generalization) from two given logged Java classes with a special attention to logging calls
- Solution: Developing an Algorithm (depicted in Figure 3) that:

- maps the source code of two LJCs to AST structures via the Eclipse JDT framework
- creates an extension of AST structures, called AUAST, to allow the insertion of variables in place of any nodes

  - determines potential candidate structural correspondences between AUAST nodes using the Jigsaw framework
  - applies some constraints to prevent the anti-unification of logging calls with anything else
  - determines the best structural correspondences between AUAST nodes to handle the problem of having multiple anti-unfiers using a greedy selection algorithm
  - constructs an anti-unifier through anti-unification of structural properties
  - develops a measure of similarity between the two AUASTs

- **Chapter 4.2 Anti-unifying a set of LJCs**
- Problem: anti-unifying a set of AUASTs of LJCs
- Solution: Developing a modified version of a hierarchical agglomerative clustering algorithm (illustrated in Figure 4) as described below:

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

  - Repeat Steps 3,4, and 5 until the similarity between closest clusters becomes below a predetermined threshold value
  - The similarity between a pair of clusters is defined as the similarity between their AUASTs
  - Determine the similarity threshold value through informal experimentation

- **Chapter 5. Evaluation**
- Two empirical studies were conducted
- **Section 5.1 Experiment 1**
- First experiment is conducted to evaluate the accuracy of our approach and tool
- It addresses the following research questions:
  - RQ1: can our tool determine the structural similarities and differences between LJCs correctly?
  - RQ2: can our tool compute the similarity between LJCs correctly?

- 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 between LJCs 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 results taken form our tool are compared with the results taken by manual investigation using the JUnit testing framework
- **Section 5.2 Experiment 2**
- Second experiment is conducted to address the following research questions:
  - RQ3: What structural similarities and differences do logged Java classes have?
  - RQ4: Is it possible to find common patterns in where logging calls do occur?
- 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
- **Section 5.2.1 Results**
- ...
- **Section 5.2.2 Lessons learned**
- ...
- **Chapter 6. Discussion**
- **Chapter 6.1 Threats to validity**
- Some potential threads to validity of our characterization study are:
  - the degree to which our sample set of software systems is a good representation of all real-world logging practices

  - the potential bias in our manual investigation to find the correct correspondences due to human errors

- **Chapter 6.2 Our tool output**
- We found that the failures of our tool happen due to:
  - the assumptions taken in developing the algorithms
  - the fundamental limitations and complexities in determining the detailed structural similarities and differences
- The are some issues that our tool is not able to handle perfectly during generalization:
  - maintaining the correct ordering of statements inside the method bodies
  - resolving all the conflicts that happen in determining the best correspondences
  - producing executable generalizations
- However, these results are still promising
- **Chapter 6.3 Theoretical foundation**
- Anti-unification and its extensions has several theoretical and practical applications:
  - analogy making [Schmidt, 2010]
  - determining lemma generation in equational inductive proofs [Burghardt, 2005]
  - detecting the construction laws for a sequence of structures [Burghardt, 2005]
- The set of equational theories in HOAUMT should be developed particularly for the structure used in each problem context
- **Chapter 7. Related Work**
- **Section 7.1 Logging**
- Logging is a systematic way of recording the software runtime information [Yuan et al., 2012]
- It has various applications, such as:
  - problem diagnosis[Jiang et al., 2009a and Jiang et al. 2009b]
  - system behavioural understanding [Fu et al., 2013 and Yaghmour et al., 2000]
  - performance diagnosis [Nagaraj et al., 2012]

  - system's security monitoring [Bishop, 1989]
  - system's recovery [Elnozahy et al. 2002]

- Yuan et al. [2012] provides a quantitative characteristic study of log messages
- Their study shows:
  - logging is a pervasive practice during software development
  - developers are not mostly satisfied with the log quality in their first attempt
  - where developers spend most of their time in modifying the log messages
- However, the focus of our study is on characterizing where developers log (not the log messages)
- **Section 7.2 Correspondence**
- Anti-unification, which is first introduced by Plotkin [1970] and Reynolds [1970], has various applications in program analysis, such as:
  - software clone detection[Bulychev and Minea, 2008]
  - generalization tasks [Cottrell et al., 2007]
  - semi-automating small scale reuse tasks [Cottrell et al., 2007]
  - recommending replacements for API migration [Bradley et al., 2014]
- These studies utilize anti-unification to detect structural correspondences between the source code fragments, however, none of them suffice to our problem context since they:
  - do not require to take constraints needed to our problem
  - do not determine the best correspondences required to our application
  - do not create the structural generalizations needed to our context
- Several other approaches have been used to find correspondences between code fragments [Baxter et al. 1998, Apiwattanapong et al., 2004, Holmes et al., 2006, Sager et al., 2006]
- However, these approaches do not determine the detailed structural similarities and differences needed in our context
- **Section 7.4 API usages patterns**
- Various data mining approaches has been used to extract API usages patterns, such as
  - association rule mining [Michail , 2000]
  - itemset mining [Li and Zhou, 2005]
  - sequential pattern mining[Xie and Pei, 2006]

- However, none of these approaches suffice to our context since they do not determine the detailed structural similarities and differences needed in our context
- **Section 7.5 Clustering**
- Clustering is an unsupervised machine mining technique that aims to organize a collection of data into clusters, such that intra-cluster similarity is maximized and the inter-cluster similarity is minimized [Karypis, 1999] [Grira et al., 2004]
- Clustering methods:
  - K-means [Hartigan et al., 1979]
    - Not a good fit to our problem since it requires to define a fixed number of clusters
  - Hierarchical clustering [Rasmussen, 1992]
    - The cut-off threshold value indicates the number of clusters we will come up with
- Agglomerative hierarchical clustering is one of the main stream clustering methods [Day, 1984] and has applications in:
  - document retrieval [Voorhees, 1986]
  - information retrieval from a search engine query log [Beeferman et al., 2000]
- **Chapter 8. Conclusion**
- Determining the detailed structural similarities and differences between source code fragments is a complex task
- It can be applied to solve several source code analysis problems, for example, characterizing logging practices
- logging is a pervasive practice and has various applications in software development and maintenance
- However, it is a challenging task for developers to understand how to use logging calls in the source code
- We have presented an approach to characterize where logging calls happen in the source code by means of structural generalizations
- We have developed a prototype tool that:
  - detects potential structural correspondences using anti-unification
  - uses several constraint to remove the correspondences that are not suited to our application
  - determines the best correspondences with the highest similarity

- constructs structural generalizations using anti-unification
- classifies the entities using a measure of similarity

- An experiment is conducted to evaluate our approach and tool
- Our experiment found that ...
- An experiment is conducted to characterize logging usage in three software systems
- In summary, our study makes the following contributions:
  - ...
  - ...
- **Section 8.1 Future Work**
- Future extensions could be applied to resolve the pitfalls of this study:
  - Data flow analysis techniques: to resolve the problem of inaccurate statement ordering
  - Further analysis: to detect and resolve all the conflicts happen in deciding the best correspondences
- However, the complexity of further analysis should be limited to keep the solution practical
- A survey can be conducted to further validate our findings
- Characterizing logging usage could be a huge step towards:
  - improving the quality of logging practices
  - developing recommendation support tools to save developers'time and effort

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

  public static void addToBus(  EBComponent comp){
    synchronized (components) {
      components.add(comp);
      copyComponents=null;
    }
  }
  public static void removeFromBus(  EBComponent comp){
    synchronized (components) {
      components.remove(comp);
      copyComponents=null;
    }
  }
  public static EBComponent[] getComponents(){
    synchronized (components) {
      if (copyComponents == null) {
        copyComponents=(EBComponent[])components.toArray(new EBComponent[components.size()]);
      }
      return copyComponents;
    }
  }
  public static void send(  EBMessage message){
    EBComponent[] comps=getComponents();
    for (int i=0; i < comps.length; i++) {
        EBComponent comp=comps[i];
        if (Debug.EB_TIMER) {
          long start=System.currentTimeMillis();
          comp.handleMessage(message);
          long time=(System.currentTimeMillis() - start);
        }
        else  comps[i].handleMessage(message);
    }
  }
}
```

Figure 1: A Java class without the usage of logging

```java
public class EditBus {
 private static ArrayList components=new ArrayList();
  private static EBComponent[] copyComponents;
  private EditBus(){
  }
  public static void addToBus(  EBComponent comp){
    synchronized (components) {
      components.add(comp);
      copyComponents=null;
    }
  }
  public static void removeFromBus(  EBComponent comp){
    synchronized (components) {
      components.remove(comp);
      copyComponents=null;
    }
  }
  public static EBComponent[] getComponents(){
    synchronized (components) {
      if (copyComponents == null) {
        copyComponents=(EBComponent[])components.toArray(new EBComponent[components.size()]);
      }
      return copyComponents;
    }
  }
  public static void send(  EBMessage message){
    Log.log(Log.DEBUG,EditBus.class,message.toString());
    EBComponent[] comps=getComponents();
    for (int i=0; i < comps.length; i++) {
      try {
        EBComponent comp=comps[i];
        if (Debug.EB_TIMER) {
          long start=System.currentTimeMillis();
          comp.handleMessage(message);
          long time=(System.currentTimeMillis() - start);
          if (time != 0) {
             Log.log(Log.DEBUG,EditBus.class,comp + ": " + time+ " ms");
          }
        }
        else  comps[i].handleMessage(message);
      }catch (Throwable t) {
         Log.log(Log.ERROR,EditBus.class,"Exception" + " while sending message on EditBus:");
      }
    }
  }
}
```

Figure 2: A Java class after the usage of logging calls

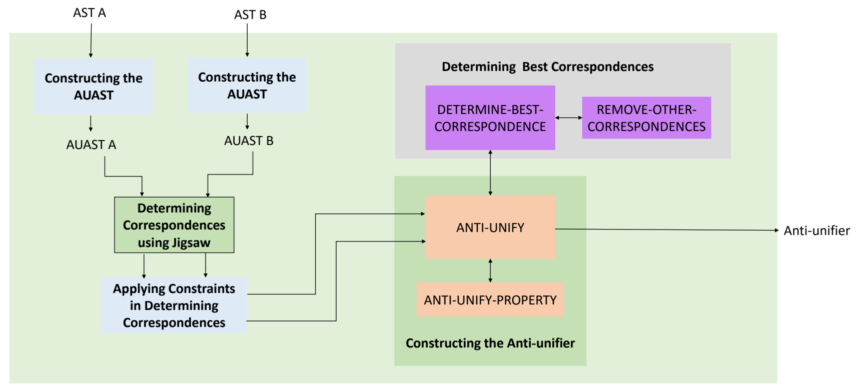
AST A
AST B
Constructing the AUAST
Constructing the AUAST
AUAST A
AUAST B
Determining Correspondences using Jigsaw
Applying Constraints in Determining Correspondences
Determining Best Correspondences
DETERMINE-BEST-CORRESPONDENCE
REMOVE-OTHER-CORRESPONDENCES
ANTI-UNIFY
ANTI-UNIFY-PROPERTY
Constructing the Anti-unifier
Anti-unifier

Figure 3: Overview of the approach for constructing an anti-unifier

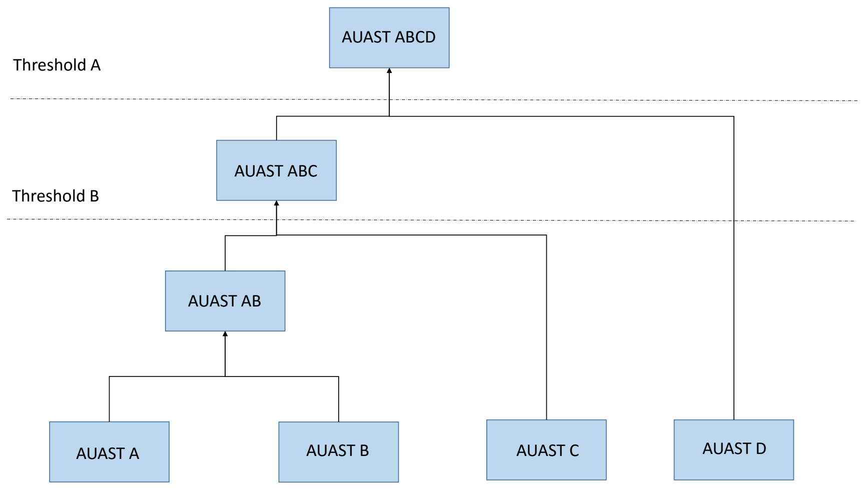

Figure 4: Anti-unification of 4 AUAST nodes using an agglomerative hierarchical clustering algorithm. Each rectangle is an indicator of a cluster. The threshold value indicates the number of clusters we will come up with.