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

- **Chapter 1. Introduction**
- **[RW: Finding commonalities and differences within the source code of a software system—or between the source code of different systems—is a complex problem to address well.]**
- **[RW: Being able to do so has several actual or potential industrial applications, for example: detecting changes within a procedure; determining the replacement for a method call when moving between program versions; collating usage patterns for an application programming interface (API); merging branches within a version control system; and so on.]**
- **[RW: As a specific application, we focus on logging.]**
- **[RW: Academically, logging has historically been considered trivial (see AOP/AOSD literature)]**
- **[RW: But several complex frameworks available, which suggests that it is both important and not straightforward]**
- **[RW: PROBLEM: We would like to understand how logging is used in practice, in a detailed way.]**
- **[RW: SOLUTION: Look automatically for commonalities and differences in its usage within a system and between systems.]**
- **[RW: The rest of this is motivational stuff.]** Developers must be able to understand software system behaviour for various development and maintenance tasks
- In practice, developers typically rely on software system's execution *logs* to understand the behaviour of large systems and to diagnose their problems
- Logging is a conventional programming practice to record an application's state and/or actions during program's execution
- The importance of logging can be seen by its various applications in:
  - problem diagnosis
  - system behavioural understanding
  - quick debugging
  - performance diagnosis
  - easy software maintenance
  - troubleshooting
- Given the importance of logging, its quality can affect the quality of an application directly

- High quality logging is not a trivial task
- Developers have to make decisions about where and what to log
- Logging should be done in an appropriate manner to be effective
  - Excessive logging:
    - can generate a lot of redundant information, masking significant ones for system log analysis
    - requires extra time and effort to write, debug, and maintain the logging code
    - can cause system resource overhead
  - Bad usage of logging can affect the application's performance
  - Insufficient logging may result in losing some necessary run-time information for software analysis
- The location of log statements has a great impact on the quality of logging since it helps developers to trace the code execution path and to identify the root causes of errors in log system analysis
- So far, little study has been conducted on characterizing logging practices in real-world applications that would assist developers to make informed decisions about where and what to log
- In this research, our focus is on the study of where developers log in practice
- **Section 1.1 Broad thesis overview**
- We aim to provide a concise description of where logging calls happen in the source code by constructing generalized structures representing common structural characteristics of where logging calls do occur
- Our approach:
  - applies a hierarchical clustering algorithm to classify the structures containing logging calls into groups using a measure of similarity
  - uses an anti-unification algorithm to construct a generalized structure representing common characteristics of all structures in each group
- Our anti-unification approach:
  - applies some constraints to avoid anti-unifying logged structures with non-logged structures
  - 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 structures containing logging calls

- Our approach has been implemented as an Eclipse plug-in, which is evaluated by conducting an empirical study on 10 sample logged Java classes and then applied across the source code of three open-source software systems that make use of logging
- Our tool extracts all Java classes containing logging calls from these systems to construct generalized structures
- Our evaluation shows ...
- **Section 1.2 Overview of related work**
- Yuan et al. [2012] provides a quantitative characteristic study of log messages on four open-source software system, however, it does not study the location of logging calls in the source code
- So far, anti-unification has been used for various applications
    - to construct a generalized correspondence view of two source code fragments (Cottrell et al. [2007])
    - to help developers to perform small-scale reuse tasks (Cottrell et al. [2008])
    - Software clone detection (Bulychev and Minea [2008])
- This study makes the first attempt to characterise where logging calls occur through the application of higher-order anti-unification modulo theories on the AST structure
- **Section 1.3 Thesis statement**
- [**RW: This will need some adjustment.**] The thesis of this work is to determine a concise description of where logging happens in real systems [**RW: You don't want to say this next part, because you never check whether developers are better supported with your solution! The thesis statement has to be a claim that is actually supported by the thesis document.**] so that developers can be better supported to make informed decisions about where to log
- **Section 1.4 Thesis Organization**
- Chapter 2: motivates the problem of characterizing the location of logging calls through an example and indicates the requirements that a potential solution must meet
- Chapter 3: provides background information on [**RW: in the intro, the point should be on what you will say and why (if the why is not self-evident)**]
    - abstract syntax trees (ASTs), which are the basic structure we will use for describing software source code

  - how ASTs are realized in the Eclipse integrated development environment, the industrial tool we will build atop
  - anti-unification and its limitations to solve our problem context
  - higher-order anti-unification modulo theories (HOAUMT) that can be applied on an extended form of the AST structure to address our problem
  - the Jigsaw framework, an existing tool for a subset of HOAUMT, which we extend to address our problem

- 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
- 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 the remaining issues.
- Chapter 8: concludes the dissertation and presents the contributions of this study and future work
- **Chapter 2. Motivational Scenario**
- [**RW: I think this fits better here. In addition to describing this background, you can give examples and show where a concise characterization could be helpful. Suggest some simple patterns (like, log the start and end of every method) and show what these might look like. Avoid ASTs at this point because you have not described them yet.**]
- 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 text message and verbosity level
  - A log text message consists of static text to describe the logged event and some optional variables related to the event
  - The verbosity level is intended to classify the severity of the logged event (Fatal, error, warn, info, and debug)
- **Chapter 3. Background**
- **Section 3.1. Abstract Syntax Tree**
- [**RW: You need to start by describing what an AST is, abstractly. Then the Eclipse stuff will make more sense.**]
- 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
  - *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
- AST nodes are subtrees of the tree structure and simple values are the leaves
- **Section 3.2. Anti-unification**
- A *term* is a set of
  - *Function symbols*:
    - that can take an unlimited number of arguments
    - are represented by identifiers starting with a lowercase letter (e.g., *f(a,b)*)
  - *Constants*: function symbols with no arguments (e.g., a, b)
  - *Variables*: are represented by identifiers starting with an uppercase letter (e.g., X, Y)
- the following are terms:
  - Y
  - a
  - *f(X, c)*
  - *f(g(X, b),Y, g(a, Z))*
- *Ground term*: a term that does not contain any variables (e.g., *f(a,b)*)
- 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
  - For example, an application of a substitution $\ominus = \; X \longleftarrow a$ on a term *f(X,b)* is defined by replacing all occurrences of variable X by term a and thus *f(X,b)* $\xleftarrow{\ominus}$ *f(a,b)*
- *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 ($X \xleftarrow{\ominus} a$)
- *Unifier*: A common instance of two given terms

- *Most General Unifier (MGU)*: U is MGU of two structures such that for all unifiers U' there exist a substitution $\ominus$ such that $U \xleftarrow{\ominus} U'$
- *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. ($X \xleftarrow{\ominus_1} a$ and $Y \xleftarrow{\ominus_2} b$)
- *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' such that $A \xleftarrow{\ominus} A'$
- 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
  - When the terms differ in function symbols, First-order anti-unification fails to preserve common details of structures, e.g., anti-unifier of structures *f(a,b)* and *h(a,b)* is X
- 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
  - e.g., anti-unifier of structures *f(a,b)* and *h(a,b)* is *X(a,b)*
- 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, e.g., for-loops and while-loops
- 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 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 the best fit to our application using an approximation technique

- **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 that is approximately the best fit to our problem
  - it does not construct an anti-unifier of two given AST structures with a focus on logging calls, which is required to our application

- **Chapter 4. Anti-unification of Logged Java Classes**

- **Goal:** Construct generalized structures describing where logging calls happen in the source code

- **Approach:** Develop an algorithm that:

- Classifies the structures containing logging calls into groups using a measure of similarity such that structures in each group has maximum similarity with each other and minimum similarity to other structures
- Abstracts commonalities of structures in each group into a generalized structure representing where logging calls do occur

- **Problem 1:** How to construct a generalization from two given logged Java classes with a special attention to logging calls?
- **Solution:** Develop an algorithm that:
  - Maps source code of the two logged Java classes to AST structures via the Eclipse JDT framework
  - Applies the higher-order anti-unification modulo theories to construct the generalized structure
    * **Subproblem:** the AST structure does not support the use of variables, which is required to construct an anti-unifier
    * **Solution:** Create an extension of AST structure, called AUAST, to allow the insertion of variables in place of any node in the tree structure, including both subtrees and leaves
  - Determines potential candidate structural correspondences between the AUAST nodes using the Jigsaw framework
  - Applies some constraints to prevent the anti-unification of logging calls with anything else, which are:
    1. A logging call should either be anti-unified with another logging call or should be anti-unified with "nothing".
    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".
  - Determines the best structural correspondences between nodes of AUASTs
    * **Subproblem:** Despite having multiple potential anti-unifiers, we need to construct one single anti-unifier from the two AUASTs that is an approximation of the best fit to our problem
    * **Solution:** Develop a greedy selection algorithm to approximate the best anti-unifier by determining the best correspondence for each node
  - Develops a measure of similarity between the two AUASTs that
    * refers to the number of identical simple values over the total number of simple values of their anti-unifier
    * calculates the similarity between simple values via the Jigsaw similarity function
- **Problem 2:** How to classify a set of AUASTs of logged Java classes?
- **Solution:** Develop 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. 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 the AUASTs of the two clusters
5. Compute the similarity between the new cluster and all remaining clusters

  - Repeat Steps 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**

  [RW: Look at Yang 2012 and try to extend their studies, then it would be more obvious what the novel contribution is here.]

- **Section 5.1 Experimental Study**
- **Section 5.2 Results**
- **Chapter 6. Evaluation**
- **Section 6.1 Logging**
- Logging is a systematic way of recording the software runtime information [Yuan et al., 2012]
- Several studies have been conducted to investigate the application of logging for various software development and maintenance tasks:
  - Jiang et al. [2009a] study the effectiveness of logging in problem diagnosis
  - Fu et al. [2013] present an approach for understanding system behaviour through contextual log analysis
  - Yaghmour et al. [2000] provide an efficient kernel-level event logging infrastructure to record and analyze system behaviour
  - Nagaraj et al., [2012] develop an automated tool to assist developer in diagnosis and correction of performance issues in distributed systems through analyzing system behaviours extracted from log data
  - Bishop [1989] proposes a formal model of systems security monitoring using logging and auditing
  - Elnozahy et al. [2002] present a survey of log-based rollback-recovery protocols
  - Jiang et al. [2009b] present an approach to automatically detect problems of load tests by mining the application execution logs

- Yuan et al. [2012] provides a quantitative [**RW: I thought you said it was qualitative**] characteristic study of log messages on four open-source software system , however, it does not study the location of logging calls in the source code [**RW: It would be smart to lean more heavily on this RW, in order to make it a clearer improvement on what they did ...**]
- **Section 6.2 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
- Burghardt and Heinz [1996] introduced anti-unification modulo equational theories to incorporate background knowledge to syntactical anti-unification which is required for some applications
- Higher-order anti-unification modulo theories is formally undecidable [Burghardt, 2005]
- Anti-unification has various applications in program analysis:
  - Bulychev and Minea [2008] suggest an anti-unification algorithm to detect software clones, however its algorithm does not suffice to our problem context since it:
    - does not require to construct a generalization
    - does not meet the requirements needed for our application to determine the best 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 its algorithm does not suffice to our problem context since it:
    - does not meet the requirements needed for our application to determine the best correspondences
  - Cottrell et al. [2008] develop Jigsaw to help developers integrate small-scale reused source code into their own code via structural correspondence through the application of higher-order antiunification modulo theories, but the algorithm does not suffice to our problem context since it:
    - does not require to construct one single generalization
    - does not require to take constraints
  - Sadi [2011] proposed an anti-unification algorithm to characterize the location of logging usages in the source code, however,

        - he has not applied anti-unification appropriately!!! [**RW: You would need to explain what that means**]

- **Section 6.3 Correspondence**
- Several approaches have been used to find correspondences between code fragments
  - Baxter et al. [1998] develops an algorithm to detect code clones in source code that uses hash functions to partition subtrees of ASTs of a program source code and then find common subtrees in the same partition through a tree comparison algorithm
  - Apiwattanapong et al. [2004] present a top-down approach to detect differences and correspondences between two versions of a Java program, through comparison of the control flow graphs created from the source code
  - Strathcona [Holmes et al., 2006] recommends relevant code snippet examples from a source code repository for the sake of helping developers to find examples of how to use an API by heuristically matching the structure of the code under development with the source code in the repository
  - Coogle [Sager et al., 2006] is developed to detect similar Java classes through converting ASTs to a normalized format and then comparing them through tree similarity algorithms
  - However, these approaches does not to determine the detailed structural correspondences needed in our context and does not allow the creation of a generalization
  - Umami [Bradley et al., 2014] presents a new approach, called Matching via Structural generalization (MSG), for API change recommendation called
    - He used the Jigsaw tool to find structural correspondences, but the algorithm does not suffice to our problem context since it:
      - does not construct a generalization
      - does not require to take constraints
- **Section 6.4 API usages patterns**
- Various data mining approaches has been used to extract API usages patterns:
  - Unordered pattern mining
    - Association rule mining: e.g., CodeWeb [Michail, 2000] uses data mining association rules to identify reuse patterns between a source code under development and a specific library
    - Itemset mining: e.g., PR-Miner [Li and Zhou, 2005] uses frequent itemset mining to extract implicit programming rules from source code and detect violations

  - Sequential pattern mining: e.g., MAPO [Xie and Pei, 2006] combines frequent subsequence mining with clustering to extract API usage patterns from the source code

- However, none of these approaches suffice to our context since they:
  - do not to determine the detailed structural correspondences needed in our context
  - do not require to construct a structural generalization

- **Section 6.5 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]

- There are various techniques to measure the distance between clusters [Rasmussen, 1992]:
  - Single linkage
  - Complete linkage
  - Average linkage
  - Centroids
  - Ward's method

- However, in our application, the distance between clusters is defined as the distance between their AUAST nodes

- **Chapter 7. Discussion**

- **Chapter 8. Conclusion**