2016

Enhanced Method Call Tree for Comprehensive Detection of Symptoms of Cross Cutting Concerns

Saleem Obaidullah Mir
Nova Southeastern University, saleem@nova.edu

Enhanced Method Call Tree for Comprehensive Detection of Symptoms of Cross Cutting Concerns

By

Saleem Obaidullah Mir

A dissertation report submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

in

Computer Science

College of Engineering and Computing

Nova Southeastern University

2016

We hereby certify that this dissertation, submitted by Saleem Mir, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

Francisco J. Mitropoulos, Ph.D. Chairperson of Dissertation Committee	Date
Renata Rand McFadden, Ph.D. Dissertation Committee Member	Date
Sumitra Mukherjee, Ph.D. Dissertation Committee Member	Date

Approved:

Ronald J. Chenail, Ph.D. Interim Dean, College of Engineering and Computing	Date

College of Engineering and Computing
Nova Southeastern University

2016

An Abstract of a Dissertation Submitted to Nova Southeastern University in Partial

Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Enhanced Method Call Tree for Comprehensive Detection of Symptoms of Cross Cutting Concerns

by

Saleem Obaidullah Mir

April 2016

Aspect oriented programming languages provide a new composition mechanism between the functional sub units as compared to earlier non aspect oriented languages. For this reason the refactoring process requires a new approach to the analysis of existing code that focuses on how the functions cross cut one another. Aspect mining is a process of studying an existing program in order to find these cross cutting functions or concerns so they may be implemented using new aspect oriented constructs and thus reduce the complexity of the existing code. One approach to the detection of these cross cutting concerns generates a method call tree that outlines the method calls made within the existing code. The call tree is then examined to find recurring patterns of methods that can be symptoms of cross cutting concerns caused by code tangling. The conducted research focused on enhancing this approach to detect and quantify cross cutting concerns that are a result of code tangling as well as code scattering. The conducted research also demonstrates how this aspect mining approach can be used to overcome the difficulties in detection caused by variations in the coding structure introduced by over time.

Acknowledgements

I would like to thank my wife for all the accommodations, support and encouragement she has provided over the years of this endeavor. She has always been a rock that I could rely on during the highs and lows of the process. I am thankful for my children’s understanding when I had to spend time on my academic work rather than spending the time with them. I would also like to thank my parents for providing me with the opportunities and background needed to undertake the effort of pursuing a doctorate degree. I would like to thank my committee members, Dr. Mitropoulos (chair), Dr. Rand McFadden, and Dr. Mukherjee, for their guidance and helpful feedback. I would especially like to thank Dr. Mitropoulos for helping me get through times where I was feeling lost during the dissertation process. Finally I would like to thank all my family, friends and colleagues who continually showed an interest and provided encouragement to me over the years of the pursuit.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Background

The description of the programming process, as described by Kiczales, Lamping, Mendhekar, Maeda, Lopes, Loingtier and Irwin (1997), is a process of functional decomposition where requirements are broken down into smaller units that represent a particular behavior of the program. These functional units are then encapsulated into programming language constructs in the form of classes, functions or procedures. In order to achieve the programs desired result these constructs are composed and coordinated in a logical sequence. Non aspect languages, referred to as general programming languages by Kiczales et al., have a single composition mechanism available for coordinating the functional units. This mechanism is in the form of method, function or procedure calls. The single mechanism forces the programmers to weave together the function calls to different programming units within the body of the code. This results in blocks of code where multiple functional units are invoked in tandem causing the functional units to cross cut each other. Kiczales et al. defines this composition of functionality where the functionality is implemented independently but executed in tandem with each other as a cross cutting concern. These woven cross cutting concerns result in code that is more complex and less readable. Moldovan and Serban (2006) also define a cross cutting concern as a feature or function of the program that is invoked in multiple places throughout the program or whose implementation is interwoven with other functions or concerns within the program. This definition of a cross cutting concern highlights two key symptoms of cross cutting concerns that can be found within a programs code. The first symptom is code scattering that is caused when references or invocations of a concern's implementation is scattered throughout the code (Moldovan & Serban, 2006). This symptom is often seen, in non AOP languages, as invocations of a method being replicated throughout various modules of the code. The second symptom of a cross cutting concern is code tangling. This condition occurs when several different concerns are interleaved with one another in the code (Moldovan & Serban, 2006). These symptoms are usually observed as recurring patterns of method invocations within the code.

The goal of AOP languages, as described by Kiczales et al (1997), is to provide programming constructs that clearly define functional units as well as mechanisms to cleanly define the compositions of those functional units without having to weave function calls within the program code. Using AOP languages concerns that cannot be cleanly encapsulated within a single distinct generalized programming procedure become candidates for an aspectized solution. For example if a particular functionality can only be achieved by calling multiple programming components interwoven together the concern is not componentized and would be a candidate for an aspectized solution. The visualization of this situation can be illustrated by an example of the observer pattern implementation as described by Hannemann and Kiczales (2002). The study shows that the concern of generating and passing events between the generators and listeners is implemented using two integration method calls. The first integration call is the registration of listeners and is usually invoked during the initialization process of the event listener. In this way the concern of registering listeners is woven into the components initialization logic. Similarly when the event generator is processing an event it invokes the notify methods of the listeners. This results in the notification logic being woven into the event handling logic of the event generating component. Hannemann and Kiczales (2002) show that by defining appropriate aspect oriented constructs the registration and notification method calls can be abstracted into an aspect and removed from the initialization and event handling code of the event generating and listener components.

Problem Statement

The aspect mining process analyzes the source code or execution data of a target program in order to identify a set of candidate cross cutting concerns (Kellens, Mens & Tonella, 2007). The difficulty in identifying representations of cross cutting concerns, as described by Marin, Deursen and Moonen (2007), is that they can be manifested in the source code in several ways. The two most common manifestations of cross cutting concerns are either implementations that are scattered throughout the code or concerns that are tangled within implementation of other functionality (Ceccato, Marin, Mens, Moonen, Tonella & Tourwe, 2005). Isolating these manifestations is a major problem during the re-engineering phase (Breu & Krinke, 2004). In order to solve this problem the research community has developed a number of different aspect mining approaches.

Given the variety of aspect mining approaches Kellens et al. (2007) conducted a study comparing several different well known aspect mining techniques. This study revealed a number of issues with the various techniques that require further attention and improvement. One such issue identified by the study was that aspect mining techniques focus on detecting one symptom of cross cutting concerns while ignoring the other. As described by the study, static aspect mining techniques that focus on analyzing the target programs source code did not include provisions for detecting cross cutting concerns that are a result of code tangling. On the other hand dynamic analysis techniques focus on analyzing execution traces of the target program and have shown the ability to detect symptoms of code tangling. However even amongst these techniques the only comprehensive aspect mining technique identified by the study was the 'Dynamic Analysis' technique developed by Bruntink, Deursen, Engelen and Tourwe (2005).

Although the dynamic analysis based techniques provide comprehensive detection capabilities Kellens et al. (2007) and Ceccato et al. (2005) point out that they require significant user effort to apply and therefore do not scale well to large applications. As an example Ceccato et al. applied the Formal Concept analysis to the JHotDraw application. This required the authors to execute 27 different use cases that were chosen after an exhaustive study of the application's documentation. As part of their conclusion Ceccato et al. noted that the effectiveness of the dynamic analysis technique is limited by the use cases chosen by the users. Based on these studies it can be concluded that for the analysis of large applications static analysis techniques would be more practical. However a comprehensive static aspect mining technique remains an open problem.

In order to overcome the limitations found with static aspect mining techniques Qu and Liu (2007) developed a new static aspect mining technique called the Method Call Tree. The Method Call Tree approach uses the target source code to generate control flow sequences that are similar to the execution traces utilized by the dynamic analysis approaches. In this way the Method Call Tree is able to analyze and detect execution patterns that are similar to the results of dynamic analysis. While the Method Call Tree technique solves one key issue found with static analysis techniques there are still certain limitations that require improvement. The first limitation is that this technique does not exhibit the ability to detect cross cutting concerns caused by code scattering. The method call tree technique also does not account for object oriented programming constructs such as polymorphic methods and class hierarchies. For example in the case where method B overrides the implementation of method C these methods may represent the same concern. If the execution patterns found within the target code include sequences invocations of a method A followed by method B as well as invocation of method A followed by method C these patterns together provide a higher degree of evidence of the existence of the same cross cutting concern. However since the method call tree does not associate inherited methods this approach considers the execution pattern of A and B to belong to a different concern than that of A and C. Due to this the evidence of the concern is dissipated and can result in the concern being overlooked. The need for accounting these object oriented constructs was identified in Marin et al. (2007) and since most of the aspect mining techniques are geared towards the analysis of object oriented languages it is vital that they account for such features.

Most of the aspect mining techniques developed by the research community have focused on detecting cross cutting concerns by identifying patterns of method calls within the target application. These approaches assume that the implementation of cross cutting concerns in the target program is consistent throughout the application. However Mens et al. (2008) states the assumption of consistency should be relaxed to account for small variations in the code since programmers have different programming styles and also because legacy code evolves over time resulting in inconsistent implementations. If these variations are not accounted for the detection process may overlook code implementations that perform the same cross cutting function but do not exactly follow the same pattern. For example the aspect mining techniques developed by Qu and Liu (2007) and Breu and Krinke (2004) attempt to identify execution relations. Breu and Krinke (2004) describe this approach as finding execution patterns that exist in the same composition within the target source code. For example an execution pattern called the inside-first-execution relation between a method A and method B denotes that the method

B is the first method called within the body of method A. In sample code this would be represented by a method definition like A(){ B(); …}. However if there is a case where the method definition of A looks like A(){C(); B();…} this implementation would not be considered as an inside-first-execution between A and B even if the invocation C does not impact the implementation of the cross cutting concern. Such instances like this example lead Mens et al. to suggest that aspect mining techniques should be able to account for variations in the implementation of cross cutting concerns.

In order to overcome the limitations discussed above the conducted research focused on enhancing the Method Call Tree aspect mining technique developed by Qu and Liu (2007) in order to resolve the short comings described above. The research focused on enhancing the method call tree algorithm to incorporate the ability to detect cross cutting concerns caused by code scattering and extrapolate the method relationships up the class hierarchy of the analyzed code. The research also addressed the issue of not considering implementation consistencies in the target code by finding patterns of executions that are similar to one another but did not match exactly.

Dissertation Goal

The conducted research aimed to enhance the existing method call tree aspect mining technique, developed by Qu and Liu (2007), in order to address three main problems. The first problem the enhanced technique addressed was the lack of ability to detect cross cutting concerns caused by code scattering. This goal was achieved by enhancing the tree generation process to include a step to record a counter for each method whenever an invocation was encountered in the target source code. In this way whenever a method call node was added to the call tree the counter for that particular method was also incremented resulting in the computation of the fan-in value for each method in the target source. The fan-in value has been shown by Marin et al. (2007) to be an effective way to identify concerns caused by code scattering. This capability was measured by applying the enhanced method call tree to the JHotDraw source code and comparing the findings against those published by Marin et al. (2007). The incorporation of the fan-in computation within the enhanced algorithm effectively addressed the lack of code scattering detection within the method call tree approach. Since the research results Qu and Liu (2007) have shown that the method call tree technique is able to generate candidate symptoms related to code tangling that are comparable to the results of the dynamic analysis technique developed by Breu and Krinke (2004), the enhanced method call tree approach combines the abilities of fan-in analysis and dynamic analysis. The combination of these abilities fulfilled a research area called out in Ceccato et al. (2007).

The second issue that the enhanced approach addressed is the lack of consideration of object oriented constructs during the analysis of the original method call tree algorithm. Marin et al. (2007) illustrated that accounting for class hierarchies and overridden methods is not a trivial process and impacts the ability to detect symptoms of code scattering. The logical conclusion is that these object oriented programming constructs impact the implementation of concerns leading to code tangling as well. Current aspect mining approaches like the dynamic analysis approach developed by Breu and Krinke (2004) as well as the rule based detection technique developed by Vidal, Abait, Marcos, Casas and Pace (2009) do not investigate the impact of object oriented programming to their respective aspect mining approaches. However the authors in both cases do point out that the impact of object oriented programming on the analysis of code tangling should be subject of future research. The conducted research attempted to account for object oriented programming constructs when analyzing the patterns of method invocations that exist within the target source code. The ultimate goal for this enhancement was to detect method relationships or tangled code that may exist across various levels of the class hierarchies.

Finally the third problem the conducted research attempted to tackle is the inability to account for slight implementation variations within the target code. According to Mens et al. (2008) current aspect mining techniques assume that the implementations of cross cutting concerns are consistent throughout the program. However given differing programming styles there may be variations in the implementations of the cross cutting concerns that make the detection of a consistent execution pattern difficult to find. The conducted research aimed to show that instead of looking for exact patterns such as method A is invoked directly before method B is invoked, the analysis can look for patterns such as method A is invoked followed directly or after a few steps by an invocation of method B. The conducted research showed that by relaxing the strictness of the patterns the enhanced method call tree approach was able to detect patterns overlooked by other aspect mining techniques. The conducted research also introduced a number of measurements that can be used to filter out inconsistent patterns that do not truly represent symptoms of code tangling.

Research Questions

1) What is the algorithm for merging the execution patterns analysis with the fan-in analysis technique?

2) What impact do class hierarchies have on the detection of symptoms of code tangling?

3) How much overhead will the extra generation of code tangling candidates affect the output of the aspect mining process?

4) What measurement can be used to determine the consistency of an execution pattern?

5) Can the control flow information stored in the call tree be used to identify any interesting patterns of usage?

Relevance and Significance

The need for aspect mining is driven by programmers looking to apply the benefits of aspect oriented programming to legacy programs written in non aspect oriented languages (Tonnella & Ceccato, 2004). Maisikeli and Mitropoulos (2012) point out that legacy systems represent a significant investment over time and must constantly adapt to changing requirements. These adaptations often lead to the constant maintenance and refactoring of the legacy code. Tribbey and Mitropoulos (2012) also point out that the evolutionary development process results in implementation of concerns getting scattered over an increasing number of modules over time. This phenomenon is evidenced by the statistical correlation between the degree of scattering of concerns in the source code and the number of defects found in the program (Tribbey & Mitropoulos, 2012). Both Maisikeli and Mitropoulos (2012) as well as Tribbey and Mitropoulos (2012) indicate that aspect oriented programming constructs would improve the reliability and maintainability of the legacy code thus lending support to refactoring legacy application code in to aspect oriented programming languages. The refactoring process required to move a legacy application to an AOP language can be both difficult and error prone. The level of difficulty increases with the size, complexity and lack of documentation of the target program (Kellens et al., 2007). These difficulties provide the motivation for the development of aspect mining tools that can automate at least some parts of the cross cutting concern detection process. The research community has introduced a number of different techniques to assist users perform the task of aspect mining but as a study by Kellens et al. shows these techniques do not comprehensively address all the challenges associated with aspect mining.

Aspect mining approaches can be divided into two broad categories based on the types of inputs used during the analysis process (Kellens et al., 2007). The first category are known as static analysis approaches and focus on analyzing the existing code to identify implementation patterns that are symptomatic of cross cutting concerns. The second category, referred to as dynamic analysis techniques, perform analysis on run time information gathered by executing the program and capturing data like the execution stack.

Studies like Kellens et al. (2007) and Ceccato et al. (2005) have shown that the static analysis techniques reviewed in those studies did not provide comprehensive detection of cross cutting concerns based on both code scattering as well as code tangling. The studies showed that static analysis techniques like fan-in analysis are able to detect concerns caused by code scattering but do not provide any mechanism for detecting symptoms of code tangling. This limitation may prevent these types of aspect mining techniques from discovering tangled code that is one of the key motivations for the development of aspect oriented programming languages as described by Kiczales et al. (1997). Similarly these two studies also point out that dynamic analysis techniques require extensive user interaction and knowledge of the target program making it difficult to apply these techniques to large programs. The described limitations of both categories prevent them from being comprehensive aspect mining approaches that can be used for large legacy systems.

A recent static aspect mining approach proposed by Qu and Liu (2009) demonstrates a static analysis technique that has the ability to detect cross cutting concerns caused by code tangling. This technique generates a method call tree and uses it to find patterns of method executions that are similar to dynamic analysis approaches. However this technique does not track the number of times methods are invoked across the target program. Therefore this method is unable to determine whether a method represents a cross cutting concern that is caused due to code scattering. This aspect mining technique also does not directly consider method hierarchies and while initial experimentation has yielded positive results the authors do note that further experimentation needs to focus on testing the algorithm using test programs with deeper inheritance hierarchies. Finally the pattern searching step of the algorithm looks for very specific patterns and does not account for the variations in implementation. This kind of pattern matching has been criticized by Mens at al. (2008) as being too strict and prone to miss interesting execution patterns that could be symptoms of code tangling.

By enhancing the method call tree approach the conducted research conclusively shows that a static aspect mining approach can meet a key aspect mining criteria of comprehensive cross cutting concern detection. The research introduced an aspect mining technique that aggregates the benefits of scalability and lower user interaction of static aspect mining techniques while providing the same detection coverage demonstrated by dynamic aspect mining techniques. Furthermore this enhanced aspect mining approach demonstrated a way to address variations in implementation that programmers may have introduced within the target source code.

Barriers and Issues

The purpose of an aspect mining technique is to identify code that represents a cross cutting concern that is a symptom of code scattering or code tangling (Mens et al., 2008). Most of the developed aspect mining techniques focus on detecting symptoms of code scattering and ignore the symptoms of code tangling (Mens et al.). Kellens et al. (2007) explains that the detection of code tangling requires some high-level information regarding the concerns implemented in the system. Techniques like dynamic analysis utilize use-case scenarios and can thus examine the methods and their compositions in the context of the concerns that are a part of the use-case. This context is difficult to provide in static analysis approaches since they focus solely on the structure of the code. This ultimately makes it difficult for them to incorporate the ability to detect code tangling.

One suggested approach to incorporate the detection of code tangling symptoms is to merge the results of different aspect mining approaches (Kellens et al., 2007). Although this approach could provide a more comprehensive aspect mining solution Mens et al. (2008) point out that merging different aspect mining techniques is not possible due to the differences in outputs of the various approaches. Another obstacle in combining aspect mining techniques, pointed out by Mens et al., is the level of subjectivity in the interpretation of the aspect mining output. Each aspect mining technique provides an output that the user has to examine in order to find the cross cutting concerns. This makes it difficult to use the output of one aspect mining technique to enhance the output of another. In this way merely merging the outputs of multiple aspect mining approaches may result in too much data for the users to sort through thus making the aspect mining process more difficult.

Aside from the lack of focus on code tangling Mens et al. (2008) discusses the need for aspect mining techniques to take into account the variability in the implementation of cross cutting concerns. The authors explain that current aspect mining techniques only identify potential cross cutting concerns if the code strictly adheres to certain anticipated patterns. For example the aspect mining approach developed by Breu and Krinke (2004) examines execution traces of a program and uses them to generate relationships like the inside-first-execution. Mens et al. postulate that legacy code may not be implemented with such uniformity and so aspect mining approaches should factor in variations during the analysis.

Assumptions, Limitations and Delimitations

The conducted research assumes that most if not all cross cutting concerns are manifested within patterns of method invocations. Certain aspect mining techniques such as the clone detection techniques developed by Bruntink, Deursen, Engelen & Tourwe (2005) detect cross cutting concerns by analyzing and identifying duplicated code. However Bruntink et al. (2005) also explain that code duplication can be caused due to improper design choices leading developers to apply a copy-paste-adapt approach while developing the code. These kinds of design choices may not necessarily be a representation of a cross cutting concern and can be resolved by a more traditional code refactoring process that does not involve an aspect oriented solution. The conducted research assumed that the target source programs had been optimized for an object oriented programming language. Therefore the research limited itself strictly to analyzing the method invocations and the detection of possible cross cutting concerns within the target program source code.

Definition of Terms

Concern - A part of the application's functionality

Cross Cutting Concern - Functionality whose implementation are interwoven within the program code, usually in the form of recurring sequences of method invocations.

Code Tangling - A symptom of a cross cutting concern where different concerns represented by different functions are interleaved with each other. These patterns of method invocations can be observed in multiple places within the code.

Code Scattering - A symptom of a cross cutting concern where a concern is used across several different parts of the application code. This symptom is often represented by a method call that is invoked by many different methods and modules across the application code.

Method Call Tree - A tree structure that illustrates a programs flow. The tree contains nodes that represent logical control flow statements like if-else conditions and loops as well as method invocations.

Fan-in Value - A numeric count of the number of times a particular method is invoked by other modules and methods within the application source code.

Candidate Seed - A method, method relationship or function that potentially indicates the presence of a cross cutting concern.

Summary

The process of aspect mining has been developed to identify cross cutting concerns that arise due to the need to intertwine functionality within a program. These cross cutting concerns manifest themselves in the form of method calls that are scattered throughout the source code or repeated patterns of interleaved method calls. Aspect oriented programming help programmers improve the readability and modularization of their code thus lowering the cost of maintaining the legacy code and making the code less prone to defects. However the process of determining candidate code for the aspect oriented refractoring however can be difficult and error prone. For this reason the research community has developed aspect mining tools and techniques to aid users identify candidates for refactoring. In order for these aspect mining tools and techniques to be effective they must provide comprehensive analysis and be easy to apply. These traits have proven to be elusive and current aspect mining techniques focus on detecting one symptom of code scattering or code tangling while ignoring the other. Additionally many of the techniques developed for the detection of code tangling require the execution of a comprehensive set of test cases and gathering run time data for the analysis thus requiring significant effort from the user.

The conducted research attempted to resolve these problems by combining two static analysis techniques that do not require any execution of the target program in order to complete the analysis. The first technique is the Fan-in analysis technique developed by Marin et al. (2007) that specializes in detecting symptoms of code scattering. The second technique is the Method Call Tree technique developed by Qu and Liu (2007) that specializes in detecting instances of code tangling. The conducted research demonstrates a process that aggregates the two different aspect mining techniques to provide a comprehensive aspect mining technique that is able to detect both symptoms of code scattering and code tangling. In addition the research has introduced a set of metrics as part of the analysis that aid the users to understand and filter the results of the analysis. In this way the conducted research introduced an aspect mining tool that provides a complete set of detection capabilities while minimizing the effort required by the users.

Chapter 2

Review of Literature

Overview

The conducted research demonstrated a way to integrate and modify existing aspect mining techniques in order to develop a comprehensive aspect mining approach. In order to understand the design considerations for the approach it was important to understand prior aspect mining techniques and research. The review of literature summarizes and highlights aspect mining techniques developed by the research community. The review highlights the approach taken by each technique along with the benefits and limitations of the approach. Since the focus of the conducted research was to combine aspect mining techniques that detect both symptoms of code scattering and code tangling the review focused on the types of symptoms the prior aspect mining techniques can detect. In order to justify the choice of aspect mining techniques chosen as the basis of the final aspect mining approach the review also includes prior studies that evaluate existing aspect mining approaches and the criteria used to compare them. These studies form the basis for the modifications implemented by the conducted research. Finally the review of literature summarizes prior research that has focused on the combination of aspect mining techniques.

The theory and research literature specific to the topic

In order to resolve the challenges of detecting cross cutting concerns in existing code the research community has developed a number of aspect mining techniques. One technique developed by Marin et al. (2007) examines the program code in order to detect cross cutting concerns caused by code scattering. This aspect mining technique relies on the computation of a fan-in value that is used to measure the extent to which a method is invoked throughout the code**.** Marin et al. (2007) note that object oriented constructs like inheritance hierarchies and overridden methods must be taken into consideration while determining the fan-in values. For this reason Marin et al. (2007) also introduced a number of rules for calculating the fan-in values for methods in programs written in object oriented languages. After applying the rules and computing the fan-in values for each method, the methods with a high fan-in value are presented to the user as potential methods that represent cross cutting concerns. In order to evaluate the aspect mining approach the technique was applied to the Tomcat, PetStore and JHotDraw source code. The experimental results generated by Marin at al. (2007) showed that filtering out methods with a fan-in value less than 10 greatly reduced the percentage of candidate concerns versus the total number of methods in the program. In order to measure the accuracy of the approach the percentage of actual concerns versus the number of candidate concerns was examined. The results also showed that for different test programs the accuracy of the fan-in analysis ranged between 51% and 87%. In conclusion the fan-in analysis aspect mining approach has shown to be a viable detection system for detecting cross cutting concerns caused by code scattering. However since the fan-in computation is concerned with only the number of invocations of a method it does not track relationships between these methods and cannot identify whether the invocations of the methods follow a particular pattern. For this reason the fan-in analysis based aspect mining approach does not provide the ability to detect symptoms of code tangling.

Moldovan and Serban (2006) proposed an augmentation to the fan-in analysis method that takes into account the fan-in value but also the number of invocations by distinct modules within the code. In addition Moldovan and Serban (2006) developed two different vector models and analyzed using cluster analysis algorithms. The first vector model used was an integer pair comprised of the fan-in value and the number of classes that invoke the method. The second vector model pairs the fan-in value along with a bit vector that represents whether or not the method is invoked by each of the other methods in the code. Using these two vector models Moldovan and Serban (2006) used various clustering algorithms to group methods together. The members of the groups formed by the cluster analysis are made up of methods that have roughly the same number of invocations from a similar number of distinct classes. In this way the cluster based approach is able to narrow the focus of the aspect mining process to the groups of methods whose distance from the zero vector is above a user specified threshold. The methods within these clusters have similar characteristics and represent candidates cross cutting concerns. Moldovan and Serban applied the cluster based aspect mining technique on JHotDraw and Carla Laffra’s implementation of Dijkstra’s algorithm. The experimentation showed that the choice of vector models had a significant impact to the cluster models and the precision of the detection although neither model emerged as a better choice overall. Based on the results of their experimentation, Moldovan and Serban (2006) concluded that the clustering based aspect mining technique was able to narrow the number of candidates that the users would need to examine to identify actual cross cutting concerns. Since the attributes used for the vector modules were related to the fan-in value and the number of distinct calls to a method across the various modules of the application, the resulting concerns detected using this approach are all symptoms of code scattering. The results of this approach did not provide any insights as to which methods or modules are invoked in conjunction with one another and therefore did not provide any code tangling detection capabilities. This limitation has been recognized by Moldovan and Serban (2006) and was recommended as an area of future research.

A different set of vector models were proposed by Tribbey and Mitropoulos (2012) that focused on the actual relationships between the methods rather than relying on aggregated values. The first model, labeled M_{FIV}, is a N x N matrix where each row represents the bit vector that represents whether or not the method is invoked by each of the other methods in the program. In this way for a matrix $[p_{ij}]$ each $p_{ij} = 1$ if method m_i is invoked by method m_j. In this way the row sums in the matrix equal the fan-in value for each method. The second model, labeled M_{FOV}, is the transpose of the M_{FIV} model such that each $p'_{ij} = 1$ if method m_j is invoked by method m_i and the row sums represent the Fan-Out values for each method. The last vector model, labeled M_{COM}, is a combination of M_{FIV} and M_{FOV} and calculated as the product of M_{FIV} and M_{FOV} divided by the diagonal value of the result if it is non zero. In addition to these base models Tribbey and Mitropoulos (2012) applied a PCA to the matrices to reduce the dimensionality of them thus creating three additional models $M^\wedge_{FIV}$ and $M^\wedge_{FOV}$ and $M^\wedge_{COM}$.

Based on the experimental results Tribbey and Mitropoulos (2012) found that the new vector models impacted the key performance measures DIV, DISP and KPREC that are used to evaluate the performance of cluster based aspect mining techniques. The DIV or measure of diversity of the clusters describes the degree of how many cross cutting concerns are found in each cluster. For a clustering based aspect mining approach the desired value for DIV is as high as possible. The DISP or dispersion measure is a measurement of how many clusters the cross cutting concern appears in. For clustering based aspect mining approaches the desired DISP value is a value that is close to 1. The KPREC measure is generated to assess the recall of the cluster model by measuring the percentage of clusters that contain actual cross cutting concerns. For a clustering based aspect mining approaches the desired value for KPREC is as high as possible. Tribbey and Mitropoulos (2012) found that the M_{COM} model provided the highest values for the DIV and DISP measurements. However the KPREC value for this was very low. Conversely the $M\wedge_{COM}$ model provided the highest KPREC value but yielded low values for the DIV and DISP measurements. Based on these findings Tribbey and Mitropoulos (2012) postulated that higher DIV and DISP measurements may actually lead to a less accurate aspect mining model. The research suggested that these measures should not be the sole measurement for a cluster based aspect mining algorithm. It should be noted that the vector models introduced by Tribbey and Mitropoulos (2012) were also based on the fan-in and dispersion of method invocations in the target code. Due to this, the aspect mining algorithm focused on identifying symptoms of code scattering but did not provide insights into execution patterns that can be symptoms of code tangling.

The clustering techniques used by the approaches discussed above utilized clustering algorithms that require the user to provide the number of expected clusters (Rand McFadden & Mitropoulos, 2012). Rand McFadden and Mitropoulos (2012) proposed using model-based clustering techniques instead of the K-means or hierarchical clustering algorithms. Their research focused on utilizing six different cluster models that automatically determine the optimum number of clusters for the dataset. The output of the new models was compared to the output of clustering algorithms used in prior research. The experimental results of the research showed that the model-based clustering algorithms reduced the number of clusters that needed to be analyzed for cross cutting concerns without significantly impacting the precision of the detection process. As with other clustering based aspect mining approaches the vector models used by Rand McFadden and Mitropoulos (2012) were based on the fan-in value and the number of executions across the various modules of the code. Therefore, as with the other clustering based aspect mining algorithms, the output of the model-based clustering technique also focuses purely on finding symptoms of code scattering but not code tangling.

Another aspect mining technique developed by Zhang and Jacobsen (2007) also factors in the fan-in and fan-out values of a method. This technique analyzed the code to generate two directed graphs. The first graph represented the programming elements that call other elements. The second graph represented the elements that are called by other elements (a reverse of the first graph). These graphs were used to rank the popularity of each element by applying the page rank algorithm. A high popularity denotes that the programming element is referenced by a large number of other elements of the code and likely indicative of a scattered concern. The approach also used the page rank algorithm to rank the elements based on their significance. The significance denotes how many other programming elements a particular element references. According to Zhang and Jacobsen (2007) a high significance rank is indicative of a core concern. The experimental results of the page rank aspect mining approach did reveal the known scattered cross cutting concerns when applied to JHotDraw. However Zhang and

Jacobsen (2007) noted that these concerns did not all have the highest popularity ranking indicating this approach has a tendency to generate a number of false positives that the user will have to analyze and reject. Although the page rank based aspect mining approach demonstrated the ability to detect symptoms of code scattering it relies solely on counting the number of times a method is invoked or invokes other methods. The algorithm does not provide any insight into the relations between methods and is not able to provide any insight into symptoms of code tangling.

Qu and Liu (2007) introduced an aspect mining technique that translates the target code into a method call tree depicting the flow of the program. The call tree is made up of nodes that represent the control flow structures (if/else and switch/case blocks) and execution branches due to method calls. Once the call tree has been generated it can be used to find relationships between methods. These relationships are represented by recurring sub trees within the entire method call tree. The experimental results presented by Qu and Liu (2007) showed that the method call tree approach has the ability to determine invocation patterns between different methods in the source code. Using this information the aspect mining algorithm has the ability to generate candidate concerns attributed to code tangling. However the method did not include a process of determining how often a method is invoked across the source code and for this reason does not detect symptoms that are purely a result of code scattering. Furthermore this approach had not been applied to any known test program and its results could not be compared to other known aspect mining techniques.

The aspect mining techniques discussed above analyzed the code and attempted to identify possible cross cutting concerns based on method invocations. An entirely different approach was taken by Bruntink, Deursen, Engelen and Tourwe (2005), who introduced an aspect mining technique that focused on identifying repeated or cloned code. This aspect mining technique uses Abstract Syntax Tree (AST) or Program Dependence Graph (PDG) utilities to convert program code into a tree format. The clone detection algorithm then attempts to find recurring sub trees that would indicate duplicated code. Bruntink et al. postulated that the instances of duplicated code can potentially represent symptoms of code scattering. Based on their experimentation Bruntink et al. found that this technique was able to identify certain cross cutting concerns such as null pointer checking and exception handling in the code of various test programs. Aspect mining based on clone detection is a technique that focuses on detecting code that is repeated throughout the program. This approach does not directly address symptoms of either code scattering or code tangling.

Another distinct type of aspect mining focuses on the semantic meaning rather than the structural nature of the code being analyzed. This approach to aspect mining has been explored by Tourwe and Mens (2004). This approach utilizes the Formal Concept Analysis technique developed as a branch of lattice theory to define aspectual views of the source code. These aspectual views, as defined by Torwe and Mens (2004), are a set of source code entities that have some structural relationship.

The first step of this process extract class names along with the methods and their parameters from the target program source code. Tourwe and Mens (2004) note that class and method names are typically constructed using words that explain the functionality that is being implemented. Based on this observation the identifier analysis approaches, like the aspectual views approach, splits the function and method names into keywords.

The unique keywords or concepts are then filtered and classified into groups. Tourwe and Mens (2004) suggest filtering out common words like 'with' and 'an' as well as a series of seven classification categories. The first category is a set of concepts where all the keywords used in the class name are used in the method names of the class. The second category groups concepts from both classes and method names where the keywords of the class name occurs in one or more of that class's method parameters. The third category groups the concepts of all classes. The fourth category groups keywords of method names that match an instance variable of the class, i.e. that are parts of accessor methods. The fifth groups keywords from all non accessor method names. The sixth category groups all keywords from methods that belong to the same class hierarchy. The seventh category defined by Tourwe and Mens (2004) defines crosscutting keywords that are part of method names within classes that do not belong to the same class hierarchy. Once the concepts have been categorized and grouped the results are displayed to the user to provide insights as common themes across the source code and determine what kinds of functionality cross cuts the program.

The aspectual view mining technique presents a tool that users can use to detect cross cutting concerns in their program source code. This technique does not suggest candidate crosscutting concerns but relies on the user to identify it based on the categories and concepts. One issue noted by Tourwe and Mens (2004) was that the basic nature of the filtering and classification process this technique generated a large noisy set of data that contained many false positives.

The identifier analysis approach to aspect mining has also been explored by Shepard, Pollock and Tourwe (2005). This approach extracts keywords from different parts of the code including comments, field names, field/method type names and method names. After filtering out common dictionary words the remaining keywords are used to identify lexical chains that are sequences of distinct words that have an equivalent semantic meaning. These lexical chains are then provided to the user who can use these lexical chains as clues during the analysis of the code. Experiments using the PetStore application code showed that this technique can be used as useful guide when examining the code. By studying the lexical chains a user can see if multiple methods have common synonyms that may indicate that a concern is being replicated by multiple methods and is a possible candidate for refactoring. This aspect mining approach can therefore indirectly support the detection of a scattered concern. This approach does not provide any real insight to the execution relationships between concerns unless the method name contains descriptions of multiple functions. Therefore this technique cannot be considered to be an approach that would be useful for detecting symptoms of code tangling. Shepard et al. (2005) also note that this aspect mining technique is very subjective and requires significant user interaction.

The aspect mining techniques discussed above are described by Kellens et al. (2007) as static mining techniques since they attempt to identify cross cutting concerns without using any run time information. Other dynamic analysis techniques have been developed that identify cross cutting concerns by analyzing run time data collected by executing the program (Kellens et al.). Breu and Krinke (2004) introduce an aspect mining approach based on analyzing the execution traces generated by executing the program using a special runtime environment that records methods invocations as they are pushed and popped from the execution stack. Breu and Krinke (2004) utilized the event traces to identify method execution relationships that may indicate a symptom of code tangling. These patterns describe a sequence of method invocations that include patterns of one method calling others as well as methods that are invoked in consistent sequences. If an execution relationship is found repeatedly in the execution traces the methods in the pattern is flagged as a potential cross cutting concern (Breu & Krinke, 2004). Based on the experimental results Breu and Krinke (2004) found that this aspect mining technique identified all expected cross cutting concerns without generating any false positives. It is important to note that the dynamic analysis aspect mining technique detects cross cutting concerns that are a result of repeated execution patterns between methods. These types of cross cutting concerns are representative of symptoms of code tangling. The dynamic analysis approach however does not detect whether a method is invoked from multiple locations within the target source code unless it is part of a commonly found execution pattern. In this way the dynamic analysis technique is not be able to detect a cross cutting concern whose only symptom is code scattering.

Tonella and Ceccato (2004) combine the dynamic analysis approach with a branch of lattice theory known as concept analysis in a new technique for aspect mining. This approach identifies methods that are commonly found in the execution traces generated by multiple use cases. The intuition behind this approach is that methods commonly executed across multiple use cases represent functions that cross cut the application. Tonella and Ceccato (2004) applied this technique to Carla Laffra's implementation of the Dijkstra algorithm. The results of the experiments matched the expected cross cutting concerns determined by independently studying several use cases. As described above the formal concept analysis based aspect mining approach determines cross cutting concerns based on how many different use cases refer to a given method. Methods that are found in execution traces of multiple use cases typically indicate that the concern represented by the method is scattered across the application. Therefore the focus of this aspect mining approach aimed at detecting symptoms of scattering. This aspect mining approach however does not examine the relationships between methods that would indicate a frequently occurring execution pattern. Therefore this aspect mining technique is not suitable for detecting symptoms of code tangling.

Another variation of the dynamic analysis of execution traces has been introduced by Vidal et al. (2009). This approach uses an association rule mining algorithm to find patterns of method pairings across different use cases. This approach treats the scenarios or use cases as transactions and the methods invoked in each execution trace as the items in the transaction. The experimental results, not published by the authors, did show that this approach has the potential to be a viable aspect mining technique (Vidal et al.). According to Vidal et al. the analysis of the association rules can reveal several interesting features of target source code. The single item frequent itemsets indicate methods that are commonly executed throughout the various use cases. These frequent methods represent candidate concerns caused due to code scattering. Secondly larger frequent itemsets in the form A → B indicate that these methods are commonly invoked in some sequence throughout the code. These kinds of relationships may indicate a symptom of code tangling. However the reliability of such rules is not as high as other code tangling detection techniques because the association rule focuses on the occurrence of the methods in the execution traces but does not determine whether the method invocations follow a specific execution pattern. Due to this reason the code tangling related candidates generated by the association rule mining approach may result in more false positives than other aspect mining approaches.

The only runtime information used by aspect mining techniques discussed above analyzes the sequence of method invocations. Maisikeli and Mitropoulos (2010) explored an aspect mining approach that analyzes software features derived through method calls, parameter sharing and method. The software features are extracted from a collection of execution traces and used to create a matrix where each row represents a vector of the six features of the method. This matrix was then used as an input to a Self Organizing Map (SOM) clustering utility to reorganize the data sets of similar methods. The resulting clusters are organized so that each cluster contains a central node surrounded by other nodes that have similar characteristics. The resulting clusters are then analyzed in a manner similar to other clustering methods described above. The central methods in the clusters are candidates of concerns caused due to code scattering. According to Maisikeli and Mitropoulos (2010) if multiple methods of the same class are found to map to the same class the pairing can represent a symptom of code tangling. However as in Vidal et al. (2009) the self organizing maps based aspect mining technique does not evaluate patterns of executions so candidates of code tangling concerns may generate more false positives than other code tangling detection processes. The experimental results generated by the study showed that this technique was able to identify all cross cutting concerns discovered by other aspect mining methods (Maisikeli & Mitropoulos, 2010) with a precision that matched or exceeded other dynamic aspect mining techniques.

Given the wide range of aspect mining approaches Ceccato, Marin, Mens, Moonen, Tonella & Tourwe (2005) analyzed three aspect mining approaches that represent common themes and methodologies found within aspect mining techniques. The approaches selected in the study were the fan-in analysis introduced by Marin et al. (2007), the formal concept analysis approach developed by Tonella and Ceccato (2004) and the identifier analysis technique of aspectualized views developed by Tourwe and Mens (2004). These techniques were applied to the JHotDraw 5.4b source code. Ceccato et al. (2005) note the aspect mining process did not have defined benchmark metrics making it impossible to develop a quantitative evaluation of these approaches. Therefore the study used a set of qualitative criteria for the comparison of the chosen techniques.

While applying the fan-in analysis technique Ceccato et al. (2005) observed that the fan-in analysis was useful for detecting concerns exemplified by three distinct situations. The first situation is when a functionality is implemented through a method so that the crosscutting behavior resides in the explicit calls to that particular method. The second situation arises when a concern is implemented using common functionality scattered through the code. These situations are detected by identifying similarities between the calling contexts. The third situation arises when a functionality is super imposed down a hierarchy of classes. In this case the concern is associated with single method but becomes a central theme across a class hierarchy.

Similarly Ceccato et al. (2005) found that the candidate concerns detected by the identifier analysis developed by Tourwe and Mens (2004), could be categorized into one of three categories. The first category included concerns that appeared like traditional aspects that did not pertain to any specific business functionality but were needed in order to implement the functionality across the application like registering listeners and persistence. The second category of concerns were concerns that were more closely related to the actual business logic of the program. These kinds of concerns, in the JHotDraw application, included drawing figures, moving objects, etc. The final category were concerns that highlighted language specific functions like iterating over collections. These types of concerns relied on or extended language specific library functions.

During the application of the Formal Concept Analysis based dynamic analysis aspect mining technique Ceccato et al. (2005) first defined 27 use-cases to be used to generate the test data needed to apply the dynamic analysis. During this phase Ceccato et al. (2005) observed two criteria for identifying cross cutting concerns. The first criteria was that the concern should be associated with a describable functionality like 'send to back' or 'handle messages'. The second criteria was that the classes involved in the functionality have a different primary responsibility that may get tangled within a use case.

After applying each aspect mining technique Ceccato et al. (2005) observed that each technique exhibited some strengths and some weaknesses when compared to one another. The fan-in analysis was found to be particularly well suited to fining concerns that depict contract enforcement or consistent behavior that is scattered throughout the code. Ceccator et al. (2005) found that these kinds of concerns were filtered out during the dynamic analysis and were not identified by the identifier analysis because their naming scheme is often unique. However Ceccato et al. (2005) do note that if a concern has a small foot print the fan-in analysis technique has a tendency to overlook it. The identifier analysis also generated a large result set that contained a significant number of false positives, making the overall detection difficult. The study also found that the candidate concerns had a tendency of being incomplete such that certain methods or functions were not considered to belong to an aspect when they should have been. The results of applying the dynamic analysis showed that this technique was able to detect some candidates missed by fan-in analysis when multiple methods represent a single concern. In these cases the concern is widely utilized but is filtered out by the fan-in analysis because each implementing method is not explicitly invoked many times. However Ceccato et al. (2005) noted that the dynamic analysis approach is limited by the scenarios or use cases chosen to generate the data for analysis and overlooks code that is not part of the executed use cases. For this reason Ceccato et al. (2005) recommended that a comprehensive cross cutting concern detection process incorporate a combination of techniques to provide complete detection capabilities.

The difficulties and problems and challenges have further been explored by Mens et al. (2008). As part of their study Mens et al. (2008) describe five major issues associated with aspect approaches. The first issue is the poor precision of many aspect mining techniques. The poor precision refers to the low percentage of relevant aspects in the set of candidate aspects generated by an aspect mining approach. This implies that the aspect mining approach generates a large set of noisy data that contains a large number of false positives. As Mens et al. (2008) describe aspect mining techniques with poor precision decrease the scalability and ease of use of the aspect mining technique. The second problem described by Mens et al. (2008) is the issue of poor recall that is tendency of the aspect mining technique overlook all of the actual aspects present in the code. The issue of poor recall results in a lack of precision in the aspect mining process. The third issue related to aspect mining techniques is the subjectivity of the process. As Mens et al. (2008) point out many of the aspect mining techniques use some user defined assumptions during the aspect mining process to filter results or categorize candidates.

These assumptions can vary from user to user making the process ambiguous to some degree. Due to this issue the same aspect mining process can yield different results when applied by different users. The fourth issue described by Mens et al. (2008) is the issue of scalability of certain aspect mining techniques. As described by Ceccato et al. (2005) dynamic analysis based aspect mining techniques require the users to identify and execute a number of use cases in order to capture the data used by the dynamic aspect mining techniques. Such issues require a significant amount of user involvement and can ultimately make using them infeasible for large complex programs. The final issue described by Men et al. (2008) is the lack of empirical validation. This issue does not necessarily impact the aspect mining techniques but is a critique of existing studies. The study points out that many aspect mining techniques have been published as proof of concepts that lack comparisons with other research. These studies point out that the aspect mining techniques demonstrate the ability to detect interesting candidates but do not provide a measure of quantitative results.

Based on the problems identified, Mens et al. (2008) also present three major root causes from which the earlier described problems originate. One of the issues identified by Mens et al. (2008) is the aspect mining techniques try to establish a general purpose approach to identifying cross cutting concerns. The study postulates that developing specific approaches tailored to certain types of cross cutting concerns may be more effective. The study also points out that aspect mining approaches rely on strong assumptions about consistency of patterns that indicate the symptoms of cross cutting concerns. In addition Mens et al. (2008) also point out that aspect mining techniques generally only generate and display evidence of a cross cutting concern while ignoring evidence that contradicts the existence of the concern. This can ultimately result in false positives. Another issue found in aspect mining techniques described in the study is that most aspect mining approaches focus on detecting either symptoms of code scattering or code tangling but not both. For this reason the aspect mining techniques can suffer from incomplete results. Additionally Mens et al. (2008) also point out that many aspect mining techniques do not use the semantic information available within the code to aid in the cross cutting concern detection process. The study points out that the semantic information can miss out on symptoms that are masked by code duplications in the target program. In addition to these issues Mens et al. (2008) also points out that the definition of what constitutes a cross cutting concern makes it difficult define and validate aspect mining processes. This issue also results in the subjectivity of the aspect mining process that ultimately affects the precision and ease of use of an aspect mining technique. The final issue with current aspect mining techniques described in the study is the inadequate representation of results. The study points out that each technique presents results in different formats and granularities. These differences make it difficult to compare and/or combine the results of different aspect mining approaches as well as making it difficult for the end user to reconcile the outputs of the aspect mining approaches.

Based on the observations in the studies performed by Ceccato et al. (2005) and Mens et al. (2008) some studies have been conducted to combine different aspect mining approaches in order to validate the results of each technique and fill in any gaps of the detection processes. One such study was performed by Ceccato, Marin, Mens, Moonen, Tonella and Tourwe (2006) as an extension of their previous study. In this study Ceccato et al. (2006) combine the results of the fan-in computation, formal concept analysis of identifiers and the formal concept analysis of execution traces to analyze the JHotDraw source code. The combination was achieved by using the fan-in analysis and dynamic analysis results to generate candidate seeds. The method names are then used to generate a list of class and method identifiers that are related to each candidate seed. These identifiers are used as a basis for the identifier analysis to determine the nearest concept for the candidate method names. The methods contained in the nearest concept are then added to the original list of candidate seeds to aggregate the results of all three aspect mining techniques. The final expanded list can then be revised and filtered to determine the final candidate concerns.

As a result of the experimentation Ceccato et al. (2006) found that using the fan-in analysis and dynamic analysis techniques to pre filter the results of the identifier analysis improved the overall scalability of the aspect mining process. In terms of seed quality the results showed that three out of four of the candidate seeds exhibited greater recall and precision while the recall and precision of the last seed decreased.

Another framework, called Timna, for combining aspect mining approaches was developed by Shepherd, Palm, Pollock and Chu-Carroll (2005). This approach combined different aspect mining techniques to generate a set of classification rules using a training program. The training is carried out by labeling known candidates in a training program and applying each aspect mining approach to the training program code. This step results in a set of rules that are specific to a particular aspect mining approach. The rules from each aspect mining technique are aggregated to generate a propositional statement to be used in the classification step later on. Once the system has been trained each individual aspect mining technique is applied to the target program. For each method the various results of each aspect mining technique is collected and checked to see if it matches a propositional statement generated during the training phase. If a match is found the method is marked as a candidate concern. If a matching propositional statement is not found the method is not considered to be a part of a cross cutting concern. In this way all the methods can be checked to determine if do or do not represent a cross cutting concern.

In order to test the Timna framework Shepherd et al. (2005) used the JHotDraw program code to train the system and used the train system to detect cross cutting concern in the PetStore program source code. Based on their experimental results Shepherd et al (2005) found that the aggregated results of fan-in analysis and cloning detection had better precision and recall than just using the fan-in computation in general. However Shepherd et al. (2005) did note that certain combinations of aspect mining techniques resulted in varying degrees of precision and recall for different kinds of candidates. Given the way the candidates were labeled it was not possible to determine what kind of candidate concerns benefited from this approach. Given this there more research is needed to evaluate the benefit of this approach.

Summary

The research community has developed a wide range of aspect mining techniques. The various approaches can be split into two broad categories. The first category of aspect mining techniques tries to locate cross cutting concerns by examining the program source code. These techniques, known as static aspect mining techniques, use a wide range of approaches that include counting method invocations, detecting patterns of repeated code and semantic deconstruction of code elements. The second category of aspect mining approaches utilizes execution traces as basis for the analysis and are known as dynamic analysis techniques. These techniques often utilize techniques such as cluster analysis or concept analysis to determine groups methods that represent common themes or concerns. Despite the differences in the approaches these various aspect mining techniques suffer from a common set of issues like being too specialized in the detection of a particular type of cross cutting concern and attempting to apply a one size fits all approach to all kinds of concerns. For this reason studies have shown that aspect mining techniques need to be improved before they can be widely used. One approach used to try to improve the aspect mining techniques is by combining different techniques during the analysis. These approaches do seem to provide some benefit in regards to the accuracy of the aspect mining process but do not entirely resolve all the issues nor do they provide consistent improvements in all cases.

Chapter 3

Methodology

Overview

The conducted research aimed to identify an aspect mining approach that combines two existing aspect mining techniques in order to aggregate their cross cutting concern detection capabilities. The conducted research started by outlining how the two computations could be merged and what improvements needed to be made to the existing algorithms. After the opportunities of combination had been analyzed the research focused on the development of an enhanced and unified algorithm that incorporated the features of the existing aspect mining algorithms. Based on this unified algorithm the research then focused on building a prototype implementation of the algorithm and used it to test the viability of the new approach. The prototype was tested by using it to analyze well known test programs like JHotDraw, Carla Laffra's implementation of the Dijkstra shortest path algorithm and the Graffiti application. The generated output was then compared to its composite approaches to verify whether the combination did indeed aggregate the abilities of its component parts.

Specific research method(s) to be employed

The conducted research aimed to create a single aspect mining approach that unifies the algorithms defined by the fan-in aspect mining approach developed by Marin et al. (2007) and the method call tree approach developed by Qu and Liu (2007). The process of defining the final algorithm took advantage of the fact that both algorithms were static analysis approaches and so both algorithms analyze the method invocations of the target program source code to generate certain metrics used to judge the presence of a concern.

The original method call tree algorithm reads the target source code and uses control flow statements and method invocations to generate a tree structure that provides a visualization of the code logic. An example of a method call tree is shown in figure 1 below.

```
methX()
{
    if(cond){
      methB();
    }else{
      methC();
   }
   methD();
}
```

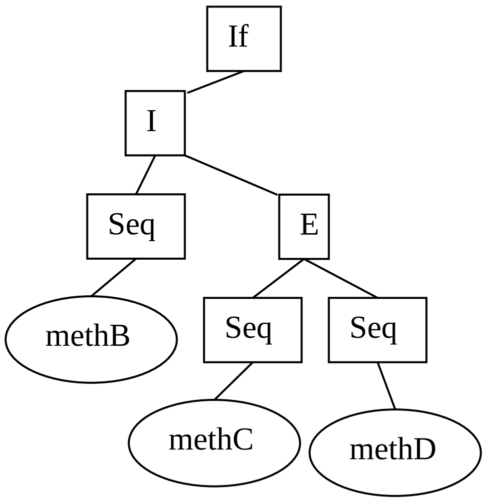

Figure 1 - Example of a Method Call Tree

As the figure shows when the method call tree algorithm reads a method call within the target code the method is added to the tree. However no further information regarding the method call is captured. After the tree is generated for the entire target program the structure of the tree is examined to identify repetitions of nodes within the branches of the trees. These repetitions indicate the presence of repeated method calls and may indicate the existence of a cross cutting concern due to code tangling.

The conducted research modified the basic method call tree algorithm by introducing two major modifications to the algorithm. The first modification is the integration of the fan-in computation as described Marin et al. (2007). The process of integrating the fan-in computation required a number counting rules to be incorporated in the method call tree algorithm. Marin et al. (2007) defines the set of rules for computing fan-in values for over ridden methods within a class hierarchy. For this reason the algorithm presented by Marin et al. (2007) includes a process to track class hierarchies during the analysis of the target source code. On the other hand Qu and Liu (2007) did not address the impact of class hierarchies and overridden methods to the generation and analysis of the method call tree. The enhanced algorithm integrated the fan-in computation by modifying the method call tree generation so that when a method definition is read the method definition and hierarchy is maintained in a look up table. This entry maintains the method name and its class hierarchy information along with a fan-in counter. After the method's meta data is captured the enhanced algorithm processed the method body as described by Qu and Liu (2007) in order to generate the set of method relationships.

The method body processing described by Qu and Liu (2007) involves reading the method body line by line examining each line for method calls and control flow constructs. When the algorithm encounters a method invocation it logs the relationship between the calling method and the method being invoked by adding the invocation node to the call tree. After the call tree is generated the tree is traversed in order to determine inside-first, inside-last, outside-before and outside-after relations between the methods as described by Breu and Krinke (2004). The algorithm developed by Qu and Liu (2007) accomplishes this by maintaining a matrix for each relationship where each cell represents a counter that increments every time the relationship between the methods in the row and column is encountered in the call tree. For example the matrix in figure 2 below would represent the matrix generated for the outside-before relationship for the sample code in figure 1.

	methB	methC	methD	methX
methB	0	0	0	0
methC	0	0	0	0
methD	1	1	0	0
methX	0	0	0	0

Figure 2 – Example of an outside-before relationship matrix

The enhanced algorithm changed the matrix so the cells maintain a list of distances between the two methods in the relationship. These lists allow the computation of the count of the relationship and also allow the computation of the consistency of the relationship. For the sample code below the derived outside-before relationship matrix generated by the enhanced algorithm is shown in figure 3.

```
methX()
{
   if(cond){
     methB();
   }else{
     methC();
  }
  methD();
}
methY()
{
   if(cond){
     methB();
   }else{
     methC();
  }
  methA();
  methD();
}
```

	methA	methB	methC	methD	methX
methA	{}	{}	{}	{}	{}
methB	{1}	{}	{}	{}	{}
methC	{1}	{}	{}	{}	{}
methD	{}	{1,2}	{1, 2}	{}	{}
methX	{}	{}	{}	{}	{}

Figure 3 – Example of an enhanced outside-before relationship matrix

In this way the algorithm calculates the count of the relationship as well as the average distance between the method calls and the variance in the distances where this relationship is found.

In addition to the altering the structure of the matrix and the data collection approach the new algorithm incorporated the hierarchical considerations introduced for computing fan-in values, introduced by Marin et al. (2007), while computing the relationships between methods. For example one such rule is that when an overriding method invocation is encountered, the fan-in counter for that method as well as its super method is incremented. The enhanced algorithm applied similar rules while determining the inside-first, inside-last, outside-before and outside-after relationships. In this way if a distance is being added to a cell in a relationship matrix the distance is also added to the cell of the methods super implementation. This process is illustrated in the example below where methA is an overridden implementation of the method superA.

```
methA extends superA
methX(){
   if(cond){
     methB();
   }else{
     methC();
  }
  methD();
}
methY(){
   if(cond){
     methB();
   }else{
     methC();
  }
  methA();
  methD();
}
```

	superA	methA	methB	methC	methD	methX
superA	{}	{}	{}	{}	{}	{}
methA	{}	{}	{}	{}	{}	{}
methB	{1}	{1}	{}	{}	{}	{}
methC	{1}	{1}	{}	{}	{}	{}
methD	{}	{}	{1,2}	{1, 2}	{}	{}
methX	{}	{}	{}	{}	{}	{}

Figure 4 – Example of an enhanced outside-before relationship matrix with overrides

As shown in the diagram every instance where the outside-before relationship for methA is captured the same relationship is added for the method superA. Once the relationship data was captured the enhanced algorithm computed the four metrics that a user could use to filter the results of the analysis and identify frequent method relationships that are likely to be instances of cross cutting concerns. The first metric computed for the method relationships was the count of the relationship that is calculated as the number of instances where the method relationship was encountered in the target code. The second metric was the average distance of the relationship that is average of all distances captured for the relationship. The third metric was the standard deviation of the distances from the average distance for the relationship. The final metric was the confidence factor for the relationship. The confidence factor for an outside relationship A outside before or after B indicates the likelihood the method A will be invoked before or after the invocation of the method B based on the relationship type. Similarly for an inside relationship the confidence factor for a relationship B inside first or last A indicated the likelihood that an implementation of method A will contain an invocation of the method B.

Instrument development and validation

In order to demonstrate the abilities of the new approach the conducted research included a prototype implementation of the enhanced algorithm. The prototype was a Java program that took a target program source code as an input, performed the aspect mining analysis described above and output the computed fan-in values and method relationships. The sample implementation focused on parsing and analysis of programs implemented in Java. Constraining the target programs to Java programs did limit the sample size that could be analyzed by the prototype to some degree. However many of the prior research tools developed by the research community including Breu and Kirnke (2004), Marin et al. (2007) and Qu and Liu (2007) also focus on the analysis of Java programs. In this way the sample implementation was able to analyze the same set of target programs that have been analyzed by the other baseline aspect mining techniques.

Validation criteria

One of the main objectives of the research was to combine the ability to detect concerns caused by code scattering and concerns caused by code tangling within a single analysis process. In order to achieve this result the research merged the fan-in computation introduced by Marin et al. (2007) with the method call tree analysis approach introduced by Qu and Liu (2007). In order to judge the success of the new algorithm the research utilized a prototype implementation to analyze the JHotDraw application source code. The results of this experiment were compared against the findings published by Marin et al. (2007) in order to ensure that results of the code scattering detection were consistent with the original fan-in computation algorithm. The success criteria for the experiment was based on whether the prototype identified the same set of high fan-in methods as were identified in Marin et al. (2007).

Similarly the validation for the detection code tangling concerns required that the new algorithm be validated against results from prior research. However since the study presented by Qu and Liu (2007) did not include detailed test results, a different study was chosen to validate the experiments. In this case the method relationships generated by the experiments were measured against the findings published by Breu and Krinke (2004). The comparison was based on the output generated by the analysis of the Graffiti source code since this code base was used as the inputs for the experiments published by Breu and Krinke (2004).

Formats for presenting results

The output of the new aspect mining technique combined the computation of fan-in values and the generation of method relationships. Therefore the output of the prototype consisted of two tables. The first table depicted by table 1 listed each method analyzed along with the computed fan-in value for the method.

Method Name	Fan-In Value

Table - 1: Sample output table for displaying computed fan-in values

The second table generated as part of the experimental results displayed the method relationships determined by the enhanced method call tree. Typically, in prior research, the method relations are displayed as a list however the output of the new algorithm also included the consistency measurements of average distance, variance and the confidence factor of the relationship. Therefore the results of the method relationships followed the structure depicted in table 2.

Relationship	Average Distance	Std Dev in Distance	Confidence Factor

Table 2 - Sample output table for displaying computed method relationships

Resource requirements

The evaluation of aspect mining algorithms rely on sample applications that have known set of cross cutting concerns in order to provide a benchmark output for comparison. The enhanced method call tree aspect mining approach was validated using the source code and prior analysis of the Carla Laffra implementation of the Dijkstra algorithm, the JHotDraw application and the Graffiti application. Therefore the research relied on obtaining the source code for these target application as experimental resources.

Summary

The conducted research aimed to fulfill its research goals by merging the fan-in computation with the computation of the method relationships generated by the method call tree analysis. The new algorithm achieved this by using the source code of the target program to generate the method call tree that was traversed in order to generate the method relationships. As part of this process whenever a method call was encountered the fan-in value for the encountered method was incremented and the method relationship was also captured. When capturing the data for the method relationships the algorithm recorded a list of distances for every instance encountered for that method relationship. This allowed the enhanced method call tree algorithm to compute the mean distance and deviation for each relationship as well as the confidence factor. In addition to capturing the relationship for the method encountered the enhanced method call tree approach also records the instance of the method relationship for the super class implementations of the method.

In order to test the abilities of the new algorithm the research included an implementation of the algorithm developed using Java. The implementation was used to analyze the source code of the JHotDraw application and the Graffiti application. The experimental results were compared against the results published by Marin et al. (2007) and Breu and Krinke (2004).

Chapter 4

Results

Data Analysis

In order to determine the effectiveness of the enhanced method call tree algorithm a set of test programs were used to generate a series of method call trees for every method within the target source code. The test programs chosen for these tests were the JHotDraw application, the Graffiti application and the Carla Laffra implementation of the Dijkstra algorithm. The enhanced method call tree algorithm was applied to each sample program to first generate the call trees that depict the sequence of method calls occurring within each functional unit within the program. These call trees then formed the basis of the raw data used for the computation steps that counted the number of times a method was invoked, that is the fan-in count of each method, as well as the method relationships between the methods within the target source code.

The first step of the data analysis transformed the method call tree into a series of execution paths. These execution were generated by traversing the call tree and listing all the method calls made within the path as shown in the sample execution trace for the method CTXCommandMenu.enable figure 5 below.

<u>Program Source Code</u>

```
public void enable(String name, boolean state) {
  for (int i = 0; i < getItemCount(); i++) {
    JMenuItem item = getItem(i);
     if (name.equals(item.getLabel())) {
      item.setEnabled(state);
      return;
    }
  }
}
```

<u>Generated execution traces</u>

```
[
 [CTXCommandMenu.getItemCount,
CTXCommandMenu.getItem,
null.getLabel, null.setEnabled, #END#],
[CTXCommandMenu.getItemCount,
CTXCommandMenu.getItem,
null.getLabel]
]
```

Figure 5 - Sample source and related execution paths

The execution paths provided a means of analyzing the method relationships as described in Breu and Krinke (2004). As shown in figure 5 above, the execution traces included both invocations of the implemented classes as well as calls made to standard library classes. In the case where a standard library calls were detected the class names were set to null within the trace output and were disregarded in any further calculations. One observation noted during the experimentation was that execution branches caused by conditional statements like 'if' statements resulted in a combinatorial explosion of the number of possible execution paths that can occur. For example the method call tree for the action method in the class Options resulted in 433 execution paths due to the high number of nested if conditions.

After generating the execution paths for a method the execution paths were used to calculate the fan-in of each method invoked within the execution paths as well as the method relationships with the invoked methods. The fan-in computation methodology employed by the modified method call tree approach differs slightly from the fan-in computation outlined by Marin et al. (2007). The difference was a result of using the execution traces as the input for the computation processes. Since a single block of code can result in multiple execution traces, due to branching instructions, this resulted in a larger number of method invocations than the count of the instances of the method within the code.

The second part of the trace analysis process was the computation of method relationships. The enhanced call tree approach captured the inside-first, inside-last, outside-before and outside-after relationships outlined by Breu and Krinke (2004). However, as outlined in the approach section, the enhanced method call tree algorithm did not limit the relationship identification to the first instance of a method invocation with the execution trace as was done in earlier approaches. For example given the execution trace for the CTXCommandMenu.enable, shown in figure 5 earlier method relationship computation approaches would identify a single inside-first method relation between CTXCommandMenu.enable and CTXCommandMenu.getItemCount. However the enhanced method call tree approach generates the relationships shown in the table 3 below

Relationship	Distance	Number of instances
CTXCommandMenu.getItemCount $\in_T$ CTXCommandMenu.enable	1	2
CTXCommandMenu.getItem $\in_T$ CTXCommandMenu.enable	2	2

Table - 3: Sample output of inside-first relationships

The distance denotes the order of method invocation within the execution trace and help form the final decision of whether the method relationship is both frequent and consistent.

Similar to inside-first and inside-last relationships the enhanced method call tree approach also analyzed the execution traces to generate outside-before and outside-after relationships. These relationships as described by Breu and Krinke (2004) denote a relationship where a method B is invoked before or after a method A within an execution trace. In a similar fashion as the inside relationships, the enhanced method call tree approach did not limit the analysis to the immediate neighbors when determining the outside-before and outside-after relationships. For example given the execution trace for the DrawApplication.loadDrawing method in the JHotDraw application

[DrawApplication.restore, Drawing.setTitle, DrawApplication.newWindow, DrawApplication.showStatus]

Table 4 shows the outside-after relationships created for the DrawApplication.restore and Drawing.setTitle methods

Relationship	Number of instances	Distance
DrawApplication.setTitle ↼ DrawApplication.restore	1	1
DrawApplication. newWindow ↼ DrawApplication.restore	1	2
DrawApplication. showStatus ↼ DrawApplication.restore	1	3
DrawApplication. newWindow ↼ DrawApplication. setTitle	1	1
DrawApplication. showStatus ↼ DrawApplication. setTitle	1	2

Table - 4: Sample output of outside-after relationships

As shown in the example in order to compute an outside-after relationship only subsequent method invocations were considered by the computation. That means that DrawApplication.restore↼Drawing.setTitle with a distance of -1, was not considered for an outside-after relationship. The reasoning for this is that such relationships are better tracked using the outside-before relationship.

The final step of the analysis used the data from the fan-in computation and method relationships to determine frequent and consistent code patterns that could indicate a cross cutting concern also referred to as a seed. The frequency of a relationship is captured by number of instances the relationship was detected within the execution traces. This indicator is a good measurement for outside-before and outside-after relationships since it describes how often the two methods can be found tangled together. However this indicator is not very relevant for inside-first and inside-last relationship since the number of instances of a relationship is always 1. Therefore when considering the inside-first and inside-last relationships the analysis focused on the number of instances a particular method occurs as the first or last method call whenever the implementation of the method is overridden.

Along with the frequency of the relationship the other important indicator for determining the viability of a seed was the consistency of the relationship. The consistency was determined by measuring the average distance recorded for all instances of a relationship. When determining the consistency of any relationship the desired average distance would be 1. In addition to the average distance the standard deviation was computed to see the degree of variability in distances for a noisy relationship. For a relationship to be consistent the standard deviation ideally would be zero. The final indicator useful for determining candidate seeds is the confidence of the relationship. The confidence of a relationship is the number of instances of the relationship/ the fan-in value for the method. If the confidence score approaches 1 that is an indication that the particular is very tightly coupled to the other method within the relationship. This would indicate that the methods are highly tangled. It should be noted however that a method can be tangled with several different concerns. For example if a method B is consistently executed after method A as well as method C the confidence of the relationships $B \leftharpoonup A$ and $B \leftharpoonup C$ with respect to B would not be close to 1.

The determination of the method relationships was enhanced to broaden the detection scheme beyond immediate neighboring method calls. This approach was found to be viable as seen in the case of the inside first relationship with the Init method calls within the GraphCanvas class. The Init method is invoked as the first method call within the GraphCanvas constructor, GraphCanvas.clear, GraphCanvas.reset, and as the second method call in the GraphCanvas.showexample method. Despite the Init method not being called as the first method in the GraphCanvas.showexample method the enhanced method call tree approach did detect the potential inside-first relationship. By computing the overall mean distance for each of the invocation instances the data clearly showed that the Init method is consistently invoked at the beginning of each method. This also makes sense given the purpose of the Init method is to initialize components prior to the drawing activities. Using the outside-before and outside-after relationships the results of the experiments using the JHotDraw application showed a consistent very tight coupling between the UndoableAdapter.setUndoable and UndoableAdapter.setRedoable methods. The results showed that the distance between the invocations had a mean distance of 1, indicating they are always adjacent to one another. The confidence factor of the relationship was also 1 showing that the methods were always invoked as a pair. This relationship is also intuitive based on the nature of the functionality of these methods. Similar relationships were observed for the FigureEnumeration.nextFigure and FigureEnumeration.hasNextFigure methods. In this case there is some variability observed since the mean distance between the two methods was found to be 1.7 however the confidence factor of 1 showed that these two methods are tightly coupled. This relationship is obvious since the hasNextFigure is a check prior to invoking the nextFigure method to obtain the next element in the enumeration.

Findings

The enhanced method call tree aspect approach successfully combined the fan-in computation with the identification of the method relationships. Moreover the combination of the two techniques resulted in a way that adds additional measurement of confidence factor that can aid in the identification of candidate seeds. Despite the combinations of two aspect mining related computations the performance of the overall algorithm was found to be very efficient. The processing times for the each of the test programs analyzed by the enhanced method call tree implementation are shown in the table 5 below.

	JHotDraw	Graffiti	Carla Laffra
Total processing time (ms)	5546	10524	1158
Class meta data collection (ms)	2674	8249	707
Tree generation time (ms)	195	178	25
Fan-in computation time (ms)	60	20	11
Method relationship computation time (ms)	1034	267	125

Table 5 - Execution times for experiments

When comparing the results of the fan-in computation experimental results showed that the fan-in values calculated by the new approach tended to be higher than those calculated by Marin et al. (2007). The table 6 below shows the differences in the high fan-in methods captured by the research and those captured by Marien et al. (2007).

Results from Marin et al. (2007)		Generated experimental results	
Class/ Method	Fan-in	Class/ Method	Fan-in
Undoable	25 (Max)	Undoable.undo	59
Storable <all methods>	22 (Total)	Storable <all methods>	29 (Total)
.willChange	25	.willChange	34
Figure.changed	36	Figure.changed	58
Figure.addFigureChangeListener	11	Figure.addFigureChangeListener	14
AbstractCommand.execute	24 15	AbstractCommand.execute	26
DecoratorFigure.containsPoint		DecoratorFigure.containsPoint	18

Table 6 - Fan-in value differences

The variation of the result was caused by the fact the enhanced method call tree approach calculates the fan-in value based on the number of method invocation within execution traces while the calculation by Marin et a. (2007) counted the fan-in by the number of occurrences of the method invocation within the source code. However when the results of the computation were compared it was found that both approaches identified a similar set of methods with high fan-in values. Since the purpose of the fan-in value is to provide a relative ranking of frequently invoked methods the results of the two approaches are consistent.

In addition to the fan-in computation, the enhanced method call tree approach demonstrated the ability to generate the set of method relationships as described in prior research such as Breu and Krinke (2004). In order to test the method relationship detection capabilities the algorithm was used to analyze the source code of the Graffiti application. The analysis results were compared to the findings described by Breu and Krinke (2004) that was also based the analysis of the Graffiti.

The first finding described by Breu and Krinke (2004) was the evidence of the logging concern. The dynamic analysis technique detected this concerned incidentally because the Graffiti application extended a standard Java API formatting class to format the log messages. The call to the log formatting was captured as part of the execution traces collected during the experimentation since it was invoked by the Java API. This concern was not detected by the enhanced call tree method since the implementation of the concern relied on the java.util.logging.Formatter standard Java API class that is called from within the java.util.logging.Logger API. As mentioned earlier the enhanced method call tree analysis focuses purely on functional concerns implemented by the target source and thus filtered out any calls to standard API methods.

As part of their analysis of concerns implemented within the Graffiti code Breu and Krinke (2004) identified the outside after relationship between the methods MainFrame.addSessionListener and the methods isSessionListener in multiple different classes. After examining the code Breu and Krinke (2004) found that the isSessionListener is defined in the interface GenericPlugin whose sub classes were detected in the execution traces. The enhanced method call tree algorithm also detected the relationship between the GenericPlugin.isSessionListener and the MainFrame.addSessionListener. In the case of the enhanced method call tree approach the directly links the relationship between the GenericPlugin interface and the MainFrame class without requiring any investigation into the code. In addition to detecting the relationship between the two methods, the enhanced method call tree approach shows that the average distance between the two method calls is 1 meaning that the relationship between the two methods is very consistent and since confidence factor of the relationship is 97% the methods are very tightly coupled together. These inferences were consistent with the explanation provided by Breu and Krinke (2004) after the examination of the Graffiti code. In a similar manner another outside after relationship discussed by Breu and Krinke (2004) is the relationship between the methods GenericPluginAdapter.getAlgorithms and the getName method of multiple different algorithm classes. After analysis of the code these algorithm classes were found to sub classes of the GenericPlugin class. This relationship was also detected by the enhanced method call tree algorithm. In the case of the enhanced method call tree algorithm the relationship between the two methods were automatically detected at the base class level. The enhanced method call tree approach found that the average distance between the getName and getAlgorithms was about two method calls. This differs from the analysis performed by Breu and Krinke (2004). Analysis of the code shows that between the getAlgorithms call and the getName there is a conditional call to processPathInformation. It is possible that this condition was never met during the testing by execution traces approach so this method call would not have been detected by Breu and Krinke (2004).

In addition to the outside before/after relationships Breu and Krinke (2004) discuss the inside first/last method relationship of the MainFrame.isSessionActive within the isEnabled method of the FileCloseAction, ViewNewAction, and RunAlgorithm, EditUndoAction and EditredoAction classes. These relationships were also detected by the enhanced method call tree algorithm for each individual class. In addition, since the isEnabled method is a defined within the base class GraffitiAction, the inside first relationship between the MainFrame.isSessionActive and the isEnabled was automatically extrapolated to the base class GraffitiAction. This relationship between the GraffitiAction and the isEnabled method was also identified by Breu and Krinke (2004) on further manual analysis of the code.

The results of the experiments show that the enhanced call tree method is able to determine relationships between methods that are comparable to dynamic aspect mining techniques. The application of these relationships in terms of detecting tangled concerns is illustrated by comparing the relationships found by the enhanced method call tree approach and the observations of the JHotDraw application made by Marin et al. (2007). As part of the analysis of the JHotDraw source code Marin at al. (2007) observed that the undo and redo concerns are tangled within the implementations of the functional procedures like the cut operation. The CutCommand class is responsible for implementing the cut functionality within the JHotDraw application and the execute method of the class contains the actual logic performed for the operation. In the JHotDraw application code the functional concerns of undo operations rely on the methods AbstractCommand.createUndoActivity and AbstractCommand.setUndoActivity in order to track changes that can later be reversed as part of the undo function itself. It is important to note that these functions themselves do not implement the logic of the undo functionality but are essential for collecting the data upon which the undo functional concern can operate. Since the undo functionality needs to maintain the history of all operations the data collection of undo concern is tangled throughout the other drawing functionality provided by the JHotDraw application. The manifestation of this tangled concerns result in the AbstractCommand.createUndoActivity, AbstractCommand.setUndoActivity and Undoable.setAffectedFigures method calls are found within the implementation of user commands such as the CutCommand.execute method and multiple methods within various painting tool implementation classes like TextTool. These manifestations of tangled concerns are illustrated by inside first or inside last method relations since these method relationships detect invocations of a tangled concerns within an implementation of some functionality. These types of relationships were observed in the enhanced method call tree algorithm output as shown in the table 7 below.

Relationship	Mean Distance	Number of instances
BorderTool.createUndoActivity∈⊤ BorderTool.action	2	1
BorderTool.createUndoActivity∈⊤ BorderTool.reverseAction	2	1
BringToFrontCommand.createUndoActivity∈⊤ BringToFrontCommand.execute	3	1
ChangeAttributeCommand.createUndoActivity∈⊤ ChangeAttributeCommand.execute	3	1
ChangeConnectionHandle.createUndoActivity∈⊤ ChangeConnectionHandle.invokeStart	3	1
ConnectionTool.createUndoActivity∈⊤ ConnectionTool.mouseDown	2	1
ConnectionTool.createUndoActivity∈⊤ ConnectionTool.mouseUp	18	2
ConnectionTool.createUndoActivity∈⊤ SplitConnectionTool.mouseDown	24	2
ConnectionTool.createUndoActivity∈⊤ Tool.mouseDown	4	1
CreationTool.createUndoActivity∈⊤ CreationTool.mouseUp	4	1
CutCommand.createUndoActivity∈⊤ CutCommand.execute	3	1
DeleteCommand.createUndoActivity∈⊤ DeleteCommand.execute	3	1
DragTracker.createUndoActivity∈⊤ DragTracker.mouseDown	6.7	3
DuplicateCommand.createUndoActivity∈⊤ DuplicateCommand.execute	3	1
FontSizeHandle.createUndoActivity∈⊤ FontSizeHandle.invokeStart	2	1
GroupCommand.createUndoActivity∈⊤ GroupCommand.execute	3	1
InsertImageCommand.createUndoActivity∈⊤ InsertImageCommand.execute	3	1
PasteCommand.createUndoActivity∈⊤ PasteCommand.execute	7	2

PolygonHandle.createUndoActivity∈⊤ PolygonHandle.invokeStart	2	1
PolygonScaleHandle.createUndoActivity∈⊤ PolygonScaleHandle.invokeStart	1	1
PolygonTool.createUndoActivity∈⊤ PolygonTool.mouseDown	7	1
PolyLineHandle.createUndoActivity∈⊤ PolyLineHandle.invokeStart	2	1
RadiusHandle.createUndoActivity∈⊤ RadiusHandle.invokeStart	2	1
ResizeHandle.createUndoActivity∈⊤ ResizeHandle.invokeStart	2	1
ScribbleTool.createUndoActivity∈⊤ ScribbleTool.mouseDown	4	1
SelectAllCommand.createUndoActivity∈⊤ SelectAllCommand.execute	3	1
SendToBackCommand.createUndoActivity∈⊤ SendToBackCommand.execute	3	1
TextAreaTool.createUndoActivity∈⊤ TextAreaTool.beginEdit	11.5	4
TextTool.createUndoActivity∈⊤ TextTool.endEdit	5	1
TriangleRotationHandle.createUndoActivity∈⊤ TriangleRotationHandle.invokeStart	2	1
UngroupCommand.createUndoActivity∈⊤ UngroupCommand.execute	3	1

Table 7 - Inside first method relationships detected for createUndoActivity

As shown by table 7 the individual number of inside first method relationships between the createUndoActivity method and the other functional methods can be quite large making it difficult to understand the concepts that these relationships actually depict. However an examination of the class hierarchies provided by the enhanced method call tree algorithm can summarize the 32 individual relationships into the following 8 method relationships listed below.

Tool.createUndoActivity ∈⊤AbstractTool.action
Tool.createUndoActivity ∈⊤Tool.mouseDown
Tool.createUndoActivity ∈⊤Tool.mouseup
Tool.createUndoActivity ∈⊤ BorderTool.reverseAction
Tool.createUndoActivity ∈⊤ TextAreaTool.beginEdit

Tool.createUndoActivity $\in_{\top}$ TextTool.endEdit
Command.createUndoActivity $\in_{\top}$ Command.execute
Handle.createUndoActivity $\in_{\top}$ Handle.invokeStart

This summarized list makes it easier to understand how the undo functionality is woven throughout the functionality of the JHotDraw application.

In addition to the tangled concerns the results of the outside before/after method relationships showed other interesting insights into the JHotDraw application. For example the one of the interesting common method relationships is the outside relationships found between the methods getUndoActivity or createUndoActivity and the setUndoActivity methods within the Handle and Command class hierarchies. Further examination of the code based on these relationships shows that the handling of the undo functionality usually follows a pattern of setUndoActivity(createUndoActivity()) or setUndoActivity(getUndoActivity()). These behaviors fall into the consistent behavior or contract enforcement classifications of cross cutting concerns described by Marin et al. (2007).

Summary

The enhanced method call tree static analysis approach was applied to the JHotDraw, Carla Laffra implementation of the Dijkstra algorithm and the Graffiti application source codes. The output of the experiments generated the method call trees for each method within the target source code. These call trees were used to generate the complete list of all execution paths the target source code could follow. The execution paths were then used to compute the fan-in for each method as well as generate a series of inside-first, inside-last, outside-before and outside-after relationships. For each of these relationships four metrics were computed namely the number of instances of the relationship, the average distance between the methods in the relationship, the standard deviation from the mean distance of both methods in the relationship and the confidence factor of the relationship. Finally these metrics were utilized to identify frequent method relationships that represent candidate seeds of cross cutting concerns.

The experiments showed that results generated by enhanced method call tree approach generated fan-in values higher than those generated by the method employed by Marin, Deursen and Moonen (2007). However the results of both approaches flagged similar methods as having high fan-in values. In addition to the fan-in computation the research also showed that it is possible for a static analysis technique to identify interesting method relationships using the four metrics described above.

The experimental results of the analysis of the Graffiti source code were compared to the findings described by Breu and Krinke (2004). When comparing the results of the approaches both approaches highlighted similar outside after and inside first/last method relationships. The only discrepancy between the two approaches was that the enhanced method call tree approach did not highlight the logging concern since the concern was implemented via standard Java API's that were excluded from the enhanced method call tree analysis. All other method relationships described by Breu and Krinke (2004) were also identified by the enhanced method call tree approach. During the comparison it was noted that many of the findings described by Breau and Krinke (2004) required an in depth review of the actual target code especially when identifying relationships spanning class hierarchies. However these hierarchical relationships were automatically highlighted within the results of the enhanced method call tree approach. The research also showed how the metrics of mean distance, number of instances and confidence could also be used to determine the relevancy of the method relationships.

Finally the research showed that the code tangling concerns discovered within the JHotDraw source code as described by Marin, Deursen and Moonen (2007) are represented by the inside first method relationships detected by the enhanced method call tree. In addition the research revealed a series of outside after relationships that depict a cross cutting concerns classified as consistent behavior and/or contract enforcement cross cutting concerns.

Chapter 5

Conclusions, Implications And Recommendations

Conclusions

One of the goals of the enhanced method call tree was to integrate the computation of fan-in analysis and the generation of method relationships into a single aspect mining algorithm. The research has successfully incorporated these two aspect mining approaches into a single algorithm by generating the fan-in values for each method in the target code as well as generating a series of inside-first, inside-last, outside-before and outside-after relationships. The experimental results and comparisons of findings with prior research has shown that the output of the new algorithm is consistent with the outputs of prior approaches namely Marin, Deursen and Moonen (2007) as well as Breau and Krinke (2004).

The second goal of the research was to incorporate the object oriented class hierarchy structure into the determination of method relationships. As shown by the experimental results the enhanced method call tree was able to use the class hierarchy to extrapolate method relationships up the class hierarchy. While comparing the experimental results with analysis performed by Breu and Krinke (2004) the utility of extrapolating these results allows the user to understand the conceptual relationships between functional components without having to refer directly to the target source code.

The final goal of the research was to incorporate the ability to find execution patterns in the target code even when the target code contained inconsistencies in implementation. The research demonstrated that some inconsistencies can be captured by expanding the method relationship detection beyond the immediate neighbors. The research also showed that the relaxation of the strict rules aids in identifying the interesting relationships. This was highlighted by the tangled concern of the undo functionality within the JHotDraw application. As shown in the experimental results there exists a clear relationship between the createUndoActivity and methods like the execute method. However the experimental results also show that the undo related functions are almost never called as the first method call in any functions implementation also that the undo related methods tend to follow a quite random distance from the beginning of the method implementations.

In addition to the main three goals of the research the research also attempted to find answers to five research questions. The first of these questions was the definition of an algorithm that incorporates the merging of the fan-in analysis and the identification of execution patterns. The approach outlined above outlines the steps of creating the execution tree for source code and traversing the tree to generate the execution paths. The research shows that these execution paths can then be used to compute the fan-in values for methods encountered in the execution path. These same execution paths can also be used to compute the method relationships between various methods in the target source code.

The second question the research attempted to explore was the impact of class hierarchies on the detection of code tangling. The research showed that extrapolating method relationships up a class hierarchy makes it easier to interpret conceptual relationships between functional areas within the target source code.

The third question explored by the research was the performance impact of the computation of the method relationships on the aspect mining process. During the generation of the experimental results the run times noted for the various processes were summarized in table 5. The table showed that the method relationship generation phase accounted for between roughly 5 to 20% of the overall execution time. However overall the enhanced method call tree analysis approach executed in an adequate timeframe in each experiment. It should be noted that the memory usage spiked during the method relationship generation phase as compared to fan-in computation phase. This spike was caused by the number of combinations of the method relationships is higher than the total number of methods in the source code. The fan-in computation phase computes a single value for each method so the memory requirements are bound by the number of methods in the target source. The method relationship computation can expands with the number of method calls within the body of the target source. The memory usage can be a limiting factor for the enhanced method call tree analysis approach and future versions will have to optimize the memory handling techniques to ensure that the process can handle the analysis of large applications.

The fourth question explored by the research was the identification of metrics that can be used to measure the consistency of a method relationship. The research demonstrated how the three metrics namely the mean distance between the methods in the relationship, the standard deviation from the mean distance of the occurrences of the method relationships and the confidence factor of the method relationships can be computed by the enhanced method call tree algorithm. The conducted research also showed how these measurements can be used to analyze the interestingness and consistency of the method relationships identified by the algorithm.

The final question explored by the conducted research was whether the control flow information stored in the call tree can be used to identify any interesting patterns in source code. As part of the analysis of the experimental results it was noted that in a number of places calls to methods like MainFrame.isSessionActive were made prior to an if condition in the Graffiti code base. However the conducted research could not discern how such information would provide any valuable insights into the target code.

Overall the research has shown that the enhanced method call tree approach was successful in achieving the goals it had set out to achieve. The research has shown that this approach can be used as a comprehensive static code analysis tool for the detection of both code scattering and code tangling cross cutting concerns. Also the enhanced method call tree approach exhibits a number of desirable traits in an aspect mining approach in that it is quick to execute, provides reasoning and metrics as part of the output that the user can leverage and does not require that the user be very familiar with the target application being analyzed.

Implications

The conducted research introduces a viable comprehensive static analysis approach for detecting cross cutting concerns. The ability to detect cross cutting concerns due to code scattering and code tangling using a static analysis approach provides a number of benefits. Static analysis techniques allow the target code to be analyzed without requiring any special run time environments as are needed by dynamic analysis techniques.

Furthermore the static analysis approach like the enhanced method call tree approach require very little prior understanding of the target source code being analyzed. This in turn makes it much faster to run the analysis and generate the results for analysis. In addition the research has also introduced metrics that provides guidelines for analyzing the results generated by the analysis. These metrics provide meaningful data points to the users helping them to fine tune and justify the design changes being made as part of the aspect refactoring exercise.

Recommendations

The conducted research has introduced an approach using an enhanced method call tree generation process for identifying candidate seeds using the fan-in computation and method relationships. The counters calculated by the computation were generated based on the number of occurrences of a method invocation within execution paths. The research identified this as being somewhat misleading since it can lead to inflated counts that cannot be directly correlated by a casual examination of the target source code. Future research should focus supplementing the execution path counts with an additional counter of distinct method bodies. This should make the generated data easier to analyze. Secondly the research did not focus on creating general guidelines for the metrics of number of instances, mean distance of methods, standard deviation of distances from the mean and the confidence of the method relations. Further research should be done to establish meaningful tolerances for these metrics to optimize the number of candidate seeds captured and precision of the candidates detected. Research can also focus on using these metrics as an input to cluster based detection algorithms as demonstrated by Maisikeli & Mitropoulos (2010).

One observation made during the research was that conceptual patterns could be discerned in the target code that did not necessarily translate into method relationships. For example in the JHotDraw source code the UndoActivity has multiple relationships with a number of methods within the various Figures classes. When observed individually each method relationship was not frequent enough to become a candidate seed. However overall there seems to be concept linking the UndoActivity and the manipulation with the various Figures classes. Further research should be conducted to incorporate a concept analysis feature similar to Tourw´e & Mens (2004) or an NLP based approach as described by Shepherd, Pollock & Tourw´e (2005).

In order to account for implementation inconsistencies within the target code the research focused on expanding the detection of method relationships beyond the next neighbor. This approach can account for variations that may be caused by methods not always being invoked within a specific sequence. However other relationships could possibly be observed if the trees of the called methods are also considered in the execution path. These types of relationships would not be detected by the shallow search employed by the current algorithm. Further research can be done to use the method relationships to generate transitive relationships that would be able to detect such potential method relationships.

Summary

The research showed that the enhance method call tree analysis approach successfully achieve its three primary goals. The first goal was achieved by successfully merging the fan-in computation with the method call tree generation algorithm. The second goal was achieved by modifying the method relationship calculation to bubble up the relationships to the super class implementation of the overridden methods. The final goal was achieved by adding the capability to determine the method relationships even if the method invocations were not called in a consistent sequence. These features showed that it was possible to develop an aspect mining approach that executes quickly, produces results that are easy to interpret and that does not require any special tools or prior knowledge of the target code in order to perform the analysis.

In order to enhance this static analysis tool further research should be performed in order to capture frequency of relationships within distinct method implementations. Other enhancements can focus on including an identifier analysis approach to the method relationships in order to determine dependent concepts or concerns within the functionality of the target code. Another enhancement should focus on using the method relationships to find transitive relationships in order to uncover dependencies obfuscated by nested method invocations. Finally further research can be performed to find optimal tolerances for the metrics generated by the enhanced method call tree approach.

Appendices

Appendix A

Prototype Implementation

1. Java source code files
2. Dependent libraries

Appendix B

Output of experimental results

1. Fan-in results for the Carla Laffra implementation of the Dijstkra algorithm
2. Method relationship results for the Carla Laffra implementation of the Dijstkra algorithm
3. Fan-in results for the JHotDraw application
4. Method relationship results for the JHotDraw application
5. Fan-in results for the Graffiti application
6. Method relationship results for the Graffiti application

3.

References

Breu, S. & Krinke, J. (2004). Aspect Mining Using Event Traces. In Proceedings of the 19th IEEE international conference on Automated software engineering. September 2004. Pages 310 - 315.

Bruntink, M., Deursen, A. V. , Engelen R. V. & Tom Tourwe T. (2005). On the Use of Clone Detection for Identifying Crosscutting Concern Code. IEEE Transactions on Software Engineering. October 2005. Pages 804-818.

Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P. & Tourwe., T. (2005). A Qualitative Comparison of Three Aspect Mining Techniques. In Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05). May 2005. Pages 13-22.

Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P. & Tourwe., T. (2006). Applying and combining three different aspect mining techniques. Software Quality Control, v.14 n.3. September 2006. Pages 209-231.

Shepherd D., Palm J., Pollock L., Chu-Carroll M. (2005). Timna: A Framework for Automatically Combining Aspect Mining Analyses. Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. 2005. Pages 184 - 193.

Hannemann, J. & Kiczales. G. (2002). Design pattern implementation in Java and aspectJ. In Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '02). November 2002. Pages 161-173.

Kellens, A., Mens, K. & Tonella, P. (2007). A survey of automated code-level aspect mining techniques. In Transactions on aspect-oriented software development IV,. Lecture Notes In Computer Science, Vol. 4640. Springer-Verlag. January 2007. Pages 143-162.

Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C. V., Maeda, C., & Mendhekar, A. (1997). Aspect-Oriented Programming. European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997. Pages 220 - 245.

Vidal, S., Abait, E. S., Marcos, C., Casas, S., & Pace, J. A. D. (2009). Aspect mining meets rule-based refactoring . Proceedings of the 1st workshop on Linking aspect technology and evolution (PLATE '09). March 2009. Pages 23-27.

Maisikeli, S. G. & Mitropoulos, F.J. (2010). Aspect mining using Self-Organizing Maps with method level dynamic software metrics as input vectors. Proceedings of the 2nd International Conference on Software Technology and Engineering (ICSTE). October 2010. Pages 212-217.

Marin, M., Deursen, A. V. & Moonen, L. (2007). Identifying Crosscutting Concerns Using Fan-In Analysis. ACM Trans. Softw. Eng. Methodol. 17, 1, Article 3. December 2007. Pages 1- 37.

Marin, M., Moonen, L. & Deuresen, A. V. (2005). A Classification of Crosscutting Concerns. Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05). September 2005. Pages 673 - 676.

Mens, K., Kellens, A. & Krinke, J. (2008). Pitfalls in Aspect Mining. In Proceedings of the 2008 15th Working Conference on Reverse Engineering (WCRE '08). October 2008. Pages 113-122.

Moldovan, G. S & Serban, G. (2006). Aspect Mining using a Vector-Space Model Based Clustering Approach. In Proceedings of Linking Aspect Technology and Evolution (LATE) Workshop. March 2006.

Qu, L. & Liu, D. (2007). Aspect Mining Using Method Call Tree. In Proceedings of the 2007 International Conference on Multimedia and Ubiquitous Engineering (MUE '07). April 2007. Pages 407-412.

Rand McFadden, R. & Mitropoulos, F. J. (2012). Aspect mining using model-based clustering. Proceedings of the 2012 Southeastcon. March 2012. Pages 1-8.

Shepard, D., Pollock, L. & Tourwe, T. (2005). Using language clues to discover crosscutting concerns. Proceedings of the 2005 workshop on Modeling and analysis of concerns in software. July 2005. Pages 1 - 6.

Tonella, P. & Ceccato, M. (2004). Aspect Mining through the Formal Concept Analysis of Execution Traces. In Proceedings of the 11th Working Conference on Reverse Engineering (WCRE '04). November 2004. Pages 112-121.

Tourwe, T. & Mens K., (2004). Mining aspectual views using formal concept analysis. In: Source Code Analysis and Manipulation Workshop (SCAM). July 2004. Pages 97 - 106.

Tribbey W. & Mitropoulos. F. J. (2012). Construction and analysis of vector space models for use in aspect mining. In Proceedings of the 50th Annual Southeast Regional Conference (ACM-SE '12). March 2012. Pages 220-225.

Zhang, C. & Jacobsen, H. (2007). Efficiently mining crosscutting concerns through random walks. In Proceedings of the 6th international conference on Aspect-oriented software development (AOSD '07). March 2007. Pages 226-238.