A Survey of Aspect Mining Tools and Techniques

Yasmin Shaikh∗
Assistant Professor,
International Institute of Professional Studies,
Devi Ahilya University, Indore (M. P.) India

Sanjay Tanwani
Professor & Head
School of Computer Science & IT,
Devi Ahilya University, Indore (M. P.) India

DOI: 10.23956/ijarcsse/V7I6/0181

Abstract—Aspect mining is a specialized reverse engineering process to identify crosscutting concerns from legacy code. Research in aspect mining is concerned with the development of concepts, principles, methods and tools to identify cross-cutting concerns from existing software systems. A comprehensive literature survey on aspect mining tools and techniques is presented in this paper. The paper contains the current state of the art about aspect mining techniques. It reports main aspect mining techniques that have been proposed in literature and are used in various aspect mining tools. A brief description of aspect mining tools is also included.

Keywords—Aspect mining, cross-cutting concern, reverse engineering, aspect mining techniques, aspect mining tool.

I. INTRODUCTION

Large-scale systems are decomposed into smaller sub-systems to manage the complexity. The decomposition is generally desired to the extent that single module contains unique well defined functionality. It is generally desired to encapsulate single functionality into single modular unit. Even with the use of object-oriented programming, system behavior such as logging, security, and persistence cannot be neatly encapsulated into a single modular unit (function/class), which leads to reduced reliability and maintainability of these systems. Such functionality that always cut-across different modular units is called crosscutting concern or aspect.

The legacy software systems developed in non-aspect oriented languages have crosscutting concerns in them. It is very difficult and error-prone to manually discover the crosscutting concerns from legacy systems. Aspect mining is the search of candidate aspects in existing systems and isolating them from the system into separately described aspects. The crosscutting concerns are candidate aspects.

A number of aspect mining techniques have already been proposed. These techniques are based on different approaches. Each technique has its own advantages and limitations. The principal symptom of existence of an aspect is code duplication and other is code scattering. Most of the existing aspect mining techniques are thus based on identifying duplicate code and scattered code. This paper presents a review of different aspect mining techniques and categories them on the basis of approach used by them to uncover aspects.

II. ASPECT MINING

Whenever any new programming paradigm is adopted, it leads to a question how to migrate existing system to that new paradigm. The same happened with aspect-oriented programming. The problem of migrating the legacy code into aspect-oriented code is decomposed into two sub problems. The first part of the problem is identification of potential candidate aspects from the existing code. The research area of aspect mining addresses the first part of problem that is identification of aspects from legacy code. The second part of problem is to transform the identified candidate aspects into actual aspects. This transformation process is referred to as aspect refactoring. Aspect mining is defined as a specialized reverse engineering process, which aims at investigating legacy systems (source code) in order to discover which parts of the system can be crosscutting concern i.e. candidate aspect. The crosscutting concerns are of two types - homogeneous concerns and heterogeneous concerns. Homogeneous concerns implement the same behavior repeatedly at different locations in a system, whereas heterogeneous concerns implement different behavior, related to the same functionality, at such locations.

III. ASPECT MINING TECHNIQUES

A number of aspect mining approaches have been proposed in literature. These approaches are mainly based on static or dynamic analysis of source code. The main aspect mining approaches are discussed in this section. The approaches discussed here give a broad view of aspect mining literature.

1) Static Code Analysis

In static code analysis, the static structure of source code is analyzed to identify the aspects from source code. One of the pioneer research papers on aspect mining techniques explored a number of research directions to advance the state of the art in aspect mining and refactoring [1]. The issues associated with different aspect mining techniques such as clone detection, slicing, dynamic analysis, and concept analysis are addressed in the paper.

In static analysis of code, natural language processing (NLP) is applied to identify crosscutting concerns from legacy code. It is assumed that the source code is implemented following proper naming conventions [2]. NLP is used to analyze
all the natural language clues that developers insert in comments and source code. All the extracted information is represented in the action-oriented identifier graph (AOIG). Such a graph models the relation between verbs and nouns using the verb-do pairs, verb and noun pairs where the verb represents an action (operation) and the noun is the object of the action. This graph has been used to build a tool supporting not only aspect mining, but also feature location and working set recovery.

Another approach of static code analysis is identifier analysis. Two major approaches involving identifier analysis based on formal concept analysis (FCA) and polymorphism are discussed here. Identifier analysis based on formal concept analysis for mining aspectual views is introduced in [3]. Design and programming patterns provide a common vocabulary shared by programmers. Identifier analysis relies on this assumption and identifies candidate seeds by grouping program entities with similar names. An aspectual view is a set of source code elements, e.g., class hierarchies, classes, and methods that are structurally related. The identifiers associated with a method or class is computed by splitting up its name based on capitals (camel-casing). For example, a method named createUndoActivity yields three identifiers create, undo and activity.

The proposed mining approach has five phases: generate elements and properties, generate the concept lattice, filter out unimportant concepts, analyze and classify remaining concepts, and finally display concepts. The proposed identifier analysis discovers interesting and meaningful aspectual views, producing many details. However, due to the lightweight technique used, it suffers from many false positives despite filtering out unimportant concepts in phase three of the analysis. False negatives can also occur due to program elements not sharing the exact same sub string, thus splitting one concept in two or more. If naming conventions are not followed properly during software development, the approach may miss the aspects from code.

In object-oriented programming, polymorphism allows methods belonging to different classes to have the same signature. Moreover, it is good practice to use intention-revealing names [4]. The methods are grouped on the basis of similar names and/or same signature. Porter stemming algorithm was applied to make sure that identifiers with the same root form (like undo and undoable) are mapped to one single representative identifier or “stem” [5]. These stems are used as FCA-attributes for the concept analysis. The FCA algorithm then groups entities with the same identifiers. When such a group contains a certain minimum number of objects (classes and methods) and the entities contained in it cut across multiple class hierarchies, the group is considered a candidate aspect seed. The only remaining (though most difficult) task is that of deciding manually whether a candidate seed is a real seed or a false positive.

Code scattering is also a symptom of crosscutting functionality. Code scattering is addressed by function/method calls. When AOP languages were not available, developers used to implement typical crosscutting concerns (such as logging or notification) in a single method that was called from many scattered places in the code. In this way they avoided tangling between the concern and the base code, but scattered calls still remain as well as a dependency from the base (non-oblivious) code to the concern code. A number of approaches have been proposed in literature that exploits the method calls to identify code scattering and thus crosscutting concerns. Two main approaches namely unique method identification and fan in analysis are discussed here.

Unique method identification is based on a simple heuristics that unique method has been used to implement an aspect mining method [6]. A unique method is defined as a method without a return value which implements a message implemented by no other method. Static analysis is used to detect all the methods that satisfy this definition. They are then sorted according to the number of times they are called and eventually altered. The unique method with higher number of method calls are considered to be crosscutting concerns. This simple approach has been able to find typical crosscutting concerns in a SmallTalk case study application.

The fan-in metric counts the number of locations from which control is passed into a module [7]. In the context of object-orientation, the module-type to which this metric is applied is the method. The fan-in of a method M has been defined as the number of distinct method bodies that can invoke M [8]. Fan-in analysis can identify crosscutting concerns such logging, tracing, pre and post-condition checks, and exception handling. The hypothesis behind this approach is that the amount of calls to a method (fan-in metric) implementing this crosscutting functionality is a good measure for the importance and scattering of the discovered concern. Fan-in analysis determines methods that are called from many different places. The methods having a high fan-in represent functionality that is used across the software system. The fan-in analysis consists of three major steps.

i. Automatic computation of the fan-in metric for all methods in the investigated system.

ii. Filtering of the results from the previous step by eliminating all methods with fan-in values below a chosen threshold, eliminating the accessor methods (whose name matches a get*/set* pattern) and eliminating utility methods (like toString() and collection manipulation methods).

iii. (Partially automated) analysis of the methods in the resulting, filtered set by exploring the callers, call sites, naming convention used, the implementation and the comments in the source code.

The result of the fan-in analysis is a set of candidate seeds, represented by methods with high fan-in. This approach proved to be able to detect most of the crosscutting concerns, but those aspects are missed that have a lower fan-in value. Clone detection techniques are also used to identify crosscutting concerns from legacy code automatically [9]. The technique has been developed with the long-term vision of developing an automatic concern miner. The experiment using clone detection was performed on an industry size component. An expert developer manually marked occurrences of four types of crosscutting concerns, namely error handling, tracing, parameter checking, and memory management. Two different clone detection tools are used; one from the category of Abstract Syntax Tree (AST)-based clone detectors,
and one based on tokenized source code representation. The study revealed that certain crosscutting concerns such as parameter checking and memory handling can be identified very well using clone detection techniques, while tracing and error handling concerns proved tricky.

2) Dynamic Code Analysis

Dynamic code analysis technique is based on the relationship between the execution traces and executed computation units [10]. If a method is executed from different computation units (e.g. classes) then such method is considered to be a crosscutting concern. Dynamic information is used for aspect mining. First, execution traces for the main functionality of a software system are generated. Then, formal concept analysis is applied to the relationship between execution traces and executed computational units (i.e., class methods). In the resulting lattice, candidate aspects are detected by determining class methods that come from multiple modules (i.e., classes) and belong to multiple use-cases. The approach is semi-automated. If no use cases for execution trace generation exist, these have to be defined. After that everything is automated up to computing the concept lattice. Interpreting the lattice and identifying concerns for refactoring into aspects is then manual again. No prior knowledge of the software is needed for this interpretation, which is an advantage over some of the other techniques.

Another aspect mining approach using event traces analyses program traces reflecting the run-time behavior of a system rather than its static structure [11]. The approach is inspired by the shopping basket analysis approach in data mining i.e. association rule mining. The program traces generated by execution of different program executions are gathered to build the data pool for aspect mining. The traces are then investigated for recurring execution patterns based on different constraints, for example, the requirement that the patterns have to exist more than once or exist in different calling contexts in the program trace.

A hybrid approach is proposed where the dynamic information of the previous approach of mining aspects from event traces is complemented with static type information such as static object types in order to remove ambiguities and thus improve on the results by reducing false positives [12]. Additionally, considering method calls rather than method executions in the traces finds more candidate aspects.

Dynamic Analysis technique has mainly two limitations. First, it is partial (i.e., not all methods involved in an aspect are retrieved), and it can determine only aspects that can be discriminated by different execution scenarios (e.g., aspects that are exercised in every program execution cannot be detected). Second, it does not deal with code that cannot be executed (e.g., code that is part of a larger framework, but is not used in a specific application).

3) Data Mining Based Aspect Mining Techniques

Data mining techniques have been applied to mine the aspects from code. This includes mining source code statically or dynamically. Clustering and association rule mining on execution traces are applied to identify crosscutting concerns [13].

Execution traces are clustered and execution traces of ordered calls are mined for association rules to determine the candidate aspects. The approach determines both the crosscutting code (advice) and the cross-cuts (point-cuts).

K-means and agglomerative clustering is used to identify crosscutting concerns [14]. Methods with high number of calling methods and calling classes are considered candidate aspects. The vector - space model is used for modeling clusters of similarities between methods. A heuristic algorithm for optimizing the set of features of clustering based aspect mining is proposed [15]. The algorithm selects important metrics from a given set of metrics called feature model by using self-organizing maps (SOM). Clustering based aspect mining techniques may produce irrelevant and redundant results. With the use of optimal set of features, accuracy of clustering concerns increase.

Clustering has been used to group together methods that belong to a crosscutting concern. Clustering starts by assigning each method to a distinct cluster. Then, clusters with the smallest distance are recursively merged together into a new cluster [16], [17]. In [16] the distance function is based on the presence of common sub strings in method names, whereas in [17] distance is evaluated using the Static Direct Invocation Relationship (SDIR) among methods: a smaller distance is associated with methods that are frequently called together, whereas a higher distance identifies those that are called together not so often.

The proposed technique applies frequent pattern mining to find the change prone files and change coupling relationships among files. In the next stage, crosscutting concerns are identified from strongly change coupled files on the basis of structural relationship among files.

4) History Based Aspect Mining Techniques

Breu et al. have developed an approach of identifying aspects from the version history [18]. It states that crosscutting functionality does not exist from the beginning. Instead, it is introduced over time. They analyzed CVS repository and identified those changes that are likely to introduce crosscutting concerns. It is assumed that two method calls that are inserted together in the same transaction are related to each other. This observation is used to mine pairs of functions that form usage patterns from version archives [19]. History-based aspect mining (HAM) identifies and ranks crosscutting concerns by analyzing where developers add code to a program [20].

A concern mining technique named COMMIT (Concern Mining using Manual Information over Time) analyzes the source code history to statistically cluster functions, variables, types, and macros that have been changed intentionally [21]. The links between the clusters represent the seed. The approach is based on clustering references that have been added or removed together.
IV. COMPARATIVE STUDY OF ASPECT MINING TECHNIQUES

Studies have been conducted to compare the main existing aspect mining techniques. Some of the important aspect mining techniques are discussed in this section. A comparative classification of aspect mining approaches is presented in [22]. The clone detection techniques such as token – based clone detection technique, program dependence graph based clone detection technique, and abstract syntax tree based clone detection techniques are based on code duplication approach. Identifier analysis based on formal concept analysis is a technique used to identify meaningful grouping of elements that have common properties. The techniques such as fan-in analysis and dynamic analysis mine source code to find symptoms of code scattering.

A qualitative comparison of three main aspect-mining techniques, namely, identifier analysis, fan-in analysis, and dynamic analysis, is presented in [23]. Identifier analysis technique tends to produces detailed results. However, these results contain too much noise (false positives). An effective filtering is required to discover the concepts and their elements that contribute to crosscutting functionality. The discovered concepts are also often incomplete, in the sense that they do not completely “cover” an aspect or crosscutting concern.

Fan-in analysis mainly addresses crosscutting concerns that are largely scattered and have a significant impact on the modularity of the system. The downside of this characteristic is that concerns with a small code footprint and thus with low fan-in values associated, will be omitted.

The three aspect mining techniques are applied in combination and experimental results are presented in [24]. All the three techniques are able to identify seeds for well-known crosscutting concerns, but interesting differences arose for other concerns. These differences are largely due to the different ways in which the techniques work. Fan-in analysis is good at identifying seeds that are largely scattered throughout the system and that involve a lot of invocations of the same method, but it cannot be used to analyze smaller applications. Identifier analysis is able to identify seeds when the associated methods have low fan-in, but only if these methods share a common lexicon. The main drawback of this technique is the large number of reported seeds that had to be inspected manually. Finally, dynamic analysis is able to find seeds in the absence of high fan-in values and common identifiers, but the technique is only partial because it relies on execution traces.

V. LIMITATIONS OF ASPECT MINING TECHNIQUES

There are a number of limitations of code analysis technique. First and foremost limitation is that they are platform-specific. Most of the existing aspect mining techniques identify crosscutting concerns form object oriented code base. But millions of lines of code written in procedural languages, especially in COBOL, exist even today and is maintained. Second limitation is poor precision [25]. Many current-day aspect mining techniques exhibit poor precision, meaning that the percentage of relevant candidate aspects in the set of all candidates reported by a given technique is relatively low. It implies that aspect mining techniques tend to return a lot of false positives, which can be detrimental to their scalability and ease-of-use. Especially for techniques that return a large number of results, this lack of precision can be problematic, since it may require significant amount of user involvement to separate the false positives from the relevant candidate aspects.

Third limitation is scalability [25]. An important property of any given aspect mining technique is its scalability. One factor that has an impact on scalability is the time-efficiency of the tool, i.e., the amount of time required for the tool to compute its results (how long it takes for the tool to run). Most tools do not seem to be problematic in this respect. Another factor contributing to scalability, however, is the amount of user involvement required for a given technique. Often, the time required for an aspect mining tool to calculate its results is negligible with respect to the amount of time required for a tool user to pre-process the tool’s input and/or post-process and analyze its output. Although the problem of user involvement holds for several known aspect mining techniques, identifier analysis is a technique that suffers in particular from this scalability issue.

Fourth limitation is lack of empirical validation of technique [25]. A good empirical validation of aspect mining techniques requires the ability to measure the precision and recall of the results, at different levels of granularity. Most of the approaches we studied do not provide an empirical validation of their results but rather provide a more incidental validation of their work. They demonstrate how particular interesting crosscutting concerns can be identified using a specific technique. While this can give an indication of the adeptness of the technique for identifying crosscutting concerns, this neither provides any quantitative information nor a sufficient basis for objective comparison of techniques.

VI. MINING VERSION ARCHIVES

Version archives have been mined by several researchers to understand software evolution and to reduce maintenance efforts. Taking into account different releases of a system, the Evolution Matrix that represents the history of classes is introduced in [26]. It analyzes different releases of system. On the basis of size metrics tracked over a number of releases a specific vocabulary to categorize classes (e.g., Pulsar, Supernova, and White Dwarf) is defined. An approach based on summarizing source code metric values of several releases to identify change prone classes is proposed [27]. The history of changes in software systems is analyzed to detect the hidden dependencies between modules [28]. The analysis is done at the file level, rather than analyzing the real code. The release and version information of software units (modules and files) as well as modification reports are considered [29]. A visualization approach to allow an engineer to quickly grasp the evolution “nature” of modules is described in [30]. The visualization allows the developer to differentiate stable files from more volatile ones with respect to change and growth trends. The concept of logical coupling is extended and a filtering mechanism is defined in [31]. Change couplings are discovered from CVS data and visualized using JGraph.
[32]. The approach takes into account the developers checking in and out files within certain periods of time and the relationship between these files. The relationships between files can discover dependencies that are difficult to detect by other means and can also point to bad code smells [33].

Two method calls that are inserted together in the same transaction are related to each other. This observation is used to mine pairs of functions that form usage patterns from version archives [19]. System-specific rules are recovered, for example, how functions interact in the source code. These rules might have been either left undocumented or changed over time along with the changes in the source code. The proposed tool analyses each version of a file in the repository and identify new function usage patterns that got introduced in subsequent versions of each file. However, this also creates huge amounts of data, which, with growing project size, will require filter mechanisms. Also, removed patterns are currently not tracked, thus stay in the list of patterns in a software project even after they cease to exist.

Data mining is applied to locate co-change patterns of arbitrary size and apply dynamic analysis to validate their patterns and identify violations [34]. The proposed tool DynaMine, implemented as Eclipse plugin, pre-processes revision history to find method calls that were inserted and stores them in a database. Then, this database is mined for usage and error patterns. The users can then select patterns and run the accordingly instrumented program for collecting dynamic data. Any pattern violations are then presented to the user in Eclipse. The tool allows for interaction as well. The user can go back, change the patterns, and re-instrument the program, rerunning the dynamic analysis. The proposed technique is based on mining co-changing files from version archives. Co-changing files and change prone files are identified using frequent pattern mining. These co-changing files are then analyzed for crosscutting concerns.

VII. ASPECT MINING TOOLS

Several automated and semi-automated aspect mining tools have been proposed, developed and are used for aspect mining. A short discussion on major tools is presented in this section. The Aspect Browser is one of the first aspect mining tools introduced. It identifies crosscutting concerns with textual-pattern matching and highlights them in color [35]. The working assumption is that programmers follow certain naming convention in naming aspects. The naming convention patterns can be represented using textual regular expression. The success or failure of tool heavily depends on programmers having followed naming conventions during development of the software system under analysis. The visual representation of the results in color is very useful and allows easy understanding of how a concern is scattered throughout the code base being analyzed.

AMT (Aspect Mining Tool) is based on a multi-modal analysis to avoid the weakness of each individual analysis by using the strength of the two i.e. text-based as well as type based analysis [39]. Text based analysis is language independent but its success heavily depends on strict naming conventions being followed. AMT therefore combines text-based and type-based analysis of source code to reduce false positives, since type-based analysis works better for objects of different types but similar names. A later extension of the tool, called AMTex tries to overcome the limitation of visualization-based aspect mining as well as dealing with industrial-sized software systems [37]. It offers more analytical functionality than the original AMT such as composition of mining activities, manages mining tasks, and cross-analyzing the mining results.

JQuery is an exploration tool and is implemented as Eclipse plugin [38]. It provides a generic browser that allows logic queries to be defined in a tool specific query language. The queries can be run on an abstract representation of a Java program’s source code, after which the results are displayed in an initial browser view. The analysis of the source code by navigation can be based on structural relationships, regular expression matches, or complex searches for structural patterns. While navigating the tree in the browser view, the user can extend the initial results by making additional queries. The results of query are added as sub trees with the elements linked to the source code for easy further investigative navigation. This has the advantage that the user does not have to switch between views or queries that help the developer to stay focused and not lose orientation when exploring the identified crosscutting concerns.

Another framework for automated aspect mining, Ophir uses a control-based comparison inspired by code clone detection [39]. It uses program dependence graphs (PDG) to discover initial candidates for refactoring. In the next step, data-based filtering is applied to eliminate undesirable refactoring candidates. The last phase of the approach is coalesce phase (grouping phase); it identifies similar candidates and coalesces these pairs into sets of similar candidates that are the final set of refactoring candidate classes.

An automatic static aspect mining based on control flow is proposed in [40]. First, the control flow graph of the program under analysis is computed. The graph is then traversed for extracting uniform and crosscutting execution relationships. An initial evaluation showed that identified crosscutting concerns cannot be realized using AO paradigm, or the detected crosscutting concern is perfectly good style. However, the results do provide interesting insights into the crosscutting behavior of the analyzed program and thus are valuable for program understanding. In addition, the technique can be used to identify crosscutting anomalies in discovered execution relation patterns.

In addition to identifying aspects, the aspect-mining tool should provide a way for representing aspects found in a legacy system [41]. A tool is proposed to represent aspects in effective manner. Two general approaches are considered. First is direct aspect storage combined with meta-data and second is mapping of aspect anatomy to the database model. The conducted evaluation revealed that the first approach is easy to implement as well as fast and efficient. The second approach has very different but equally important strengths; it retains the aspect representation and enables fine-grained aspect mining. Thus, the authors suggest a hybrid approach combining both, direct aspect storage coupled with meta-data together with mapping aspect anatomy to the database model.
FEAT (Feature Exploration and Analysis Tool) is also an aspect-mining (and general software exploration) tool and is implemented as Eclipse plugin [42]. It visualizes concerns in a system using concern graphs. A concern graph abstracts a concern’s implementation details by storing the structure implementing that concern. Thus, it documents explicitly the relationships between the different concern elements. A concern graph’s nodes represent classes, fields, and methods; its edges represent different kinds of relations between the nodes. FEAT is query-based, with the query process being interactive; the user can submit additional queries based on the results. However, this requires a starting point for the analysis, which sometimes has to be found by trial and error. The difficulty to find such a point increases if the software system to be analyzed is unknown or large, both quite common and realistic situations. This means that domain knowledge of the software system need to be acquired before analyzing the system with FEAT.

VIII. CONCLUSION

The paper presents a comprehensive review of aspect mining techniques. Most of the aspect mining techniques are based on static or dynamic program analysis program. The problem with these approaches is their scalability. There is a need of some aspect mining technique that is not dependent on the static or dynamic analysis of code. The identification of crosscutting concerns should enable the programmer to maintain the code easily and efficiently. Millions of lines of code exist in legacy systems. Practically it is not possible to refactor the whole code into aspect oriented one. But the crosscutting code can be refactored (if possible) to reduce the degree of code scattering and code tangling in the system. A number of aspect mining tools have been proposed and developed but they are semi-automatic. Although progresses are continuously been made in the field of aspect mining techniques, there are many issues that needs to be resolved and much research has to be done.

REFERENCES

[21] B. Adams, Z.M. Jiang, and A E. Hassan, Identifying Crosscutting Concerns using Historical Code Changes,