Automated Aspect Recommendation through Clustering-Based Fan-in Analysis

Danfeng Zhang, Yao Guo*, Xiangqun Chen
Key laboratory of High Confidence Software Technologies (Ministry of Education)
Institute of Software, School of Electronics Engineering and Computer Science, Peking University
zhdf@os.pku.edu.cn, {yaoguo, cherry}@sei.pku.edu.cn

Abstract

Identifying code implementing a crosscutting concern (CCC) automatically can benefit the maintainability and evolvability of the application. Although many approaches have been proposed to identify potential aspects, a lot of manual work is typically required before these candidates can be converted into refactorable aspects.

In this paper, we propose a new aspect mining approach, called Clustering-Based Fan-in Analysis *(CBFA), to recommend aspect candidates in the form of method clusters, instead of single methods. CBFA uses a new lexical based clustering approach to identify method clusters and rank the clusters using a new ranking metric called* cluster fan-in. *Experiments on Linux and JHotDraw show that CBFA can provide accurate recommendations while improving aspect mining coverage significantly compared to other state-of-the-art mining approaches.*

1. Introduction

Since its introduction in the 1990s, Aspect-Oriented Programming (AOP) [1] has enhanced the maintenance and evolution of software by separating concerns into modules. The whole process of evolving legacy software into aspect-oriented systems can be divided into two steps: *aspect mining* (manually or automatically) to identify potential aspects in legacy software, and *refactoring* to encapsulate these aspect candidates into aspect-oriented (AO) code.

Many aspect mining approaches [2]–[9] have been proposed to identify aspect candidates. As not all the candidates identified by these tools are "true" candidates[1], we first need to filter out the "fake" ones from the mining results before these candidates can be converted into refactorable aspects.

* *Corresponding author.*

1. In this paper, "true" candidates are referred to those can be encapsulated into meaningful aspects during refactoring, while the rest are referred as "fake" candidates.

The quality of aspect recommendation plays an important role in alleviating manual work in identifying these "fake" candidates. Many current mining approaches try to identify potential aspects through single occurrences, thus, a lot of manual work is required to combine these methods into meaningful aspects, each of them normally contains multiple methods related to the same concern. Although some mining approaches can automatically group the aspect candidates, they tend to miss groups with a relatively small size due to the lack of proper recommendation criteria, as we will show later.

Our motivation is originated from a widely used aspect mining approach, fan-in analysis [4], which has been adopted in many studies [10], [11]. The basic idea behind it is that a method called by many other methods is more likely to be a crosscutting concern (CCC). Although the approach can successfully identify many potential aspect candidates, it faces the following main limitations:

- It uses a fan-in threshold to filter out methods with low fan-in value, which, as they claimed, are not likely to be aspects. However, many true candidates might have very low thresholds, which means they will be filtered out by fan-in analysis.
- Fan-in analysis only provides a set of unrelated methods with high fan-in's, without any information about which concern each candidate should belong to. Those candidates have to be partitioned into different groups manually during refactoring to create meaningful aspects, which is both time consuming and error-prone.
- The choice of fan-in threshold is application specific. As reported in [10], [12], choosing a suitable threshold has a significant impact on the mining performance of fan-in analysis.

In this paper, we propose a new approach called Clustering-Based Fan-in Analysis (CBFA), which takes advantage of clustering [13] to automatically divide the methods into separate clusters, where each cluster represents a meaningful set of methods related to the same concern. Identifying aspect candidates as clusters removes the efforts required to combine single aspect candidates into refactorable aspects. Identifying clusters rather than single methods can also improve the coverage of fan-in analysis because methods with low fan-in's can be identified together with related high fan-in ones. We also define a new ranking metric, *cluster fan-in*, which is used to rank the method clusters automatically.

This approach is applicable regardless of specific programming languages. We evaluated it on two completely different systems, *C*-based *Linux*, and a *Java* system, *JHotDraw*[2], which has been evaluated in many previous aspect mining research work [4], [10], [12]. Through detailed evaluation of several CCCs in Linux and JHotDraw, we demonstrate that CBFA can provide accurate recommendations while improving aspect mining coverage significantly compared to other state-of-the-art mining approaches.

This paper makes the following main contributions:

- By taking a lexical based clustering approach to group related concern occurrences together, CBFA achieves two important goals compared to methods identifying single occurrences individually: improving the mining coverage and reduce the cost of aspect construction during refactoring.
- We propose using a new ranking metric, *cluster fan-in*, to recommend aspect candidates in order of their significance, instead of using a cutting threshold (such as cluster size) to filter out less possible candidates, as used in most mining approaches based on grouping or clustering.
- CBFA is applicable to both *C* and *Java*. Through experiments with two popular real-life software systems, we demonstrate that CBFA displays good performance regardless of specific languages.

The paper has been structured as follows. In the next section, we present an example of applying fan-in analysis to *Linux*. In section 3, we present the proposed CBFA approach in detail. We show the performance of this new approach in section 4, and discuss some open issues in section 5. Related work will be compared in section 6. We conclude our work in section 7.

2. An Aspect Mining Example

First, we use a relatively simple aspect mining example to explain the main problems with current approaches and introduce key motivations behind our proposed approach.

One of the very important crosscutting concerns in *Linux*, called synchronization, is responsible for handling the synchronization tasks between processes and threads in the system. Synchronization has been studied in many aspect mining research, such as in *PURE* [14]. As pointed out in their work, encapsulating these code can greatly improve the architectural flexibility for an operating system.

2. www.jhotdraw.org

Table 1. Synchronization functions in *Linux*

Mechanisms	Related functions
atomic operation (11)	**ATOMIC_INIT atomic_read atomic_set** *atomic_add atomic_sub atomic_dec atomic_add_negative atomic_sub_and_test atomic_inc atomic_dec_and_test atomic_inc_and_test*
spin lock (11)	*spin_lock spin_trylock spin_unlock* **spin_lock_init spin_is_locked spin_lock_irqsave spin_lock_irq spin_lock_irqrestore spin_unlock_irq spin_unlock_irqsave spin_unlock_irqrestore**
read/write spin lock (15)	*read_lock write_lock* **read_unlock write_unlock read_lock_irq read_lock_irqsave read_unlock_irq read_unlock_irqstore write_lock_irq write_lock_irqsave write_unlock_irq write_unlock_irqrestore write_trylock rw_lock_init rw_is_locked**
big kernel lock (5)	*lock_kernel unlock_kernel* **kernel_locked release_kernel_lock reacquire_kernel_lock**

Many synchronization mechanisms are used in *Linux* (we use *Linux* 2.4.18 here), such as atomic operation, spin lock, read/write spin lock, and big kernel lock. Functions related to these mechanisms are summarized in Table 1 as listed in [15], where the bolded functions are function-like macros[3], and the others are inlined functions. The call sites of these function-like macros or inlined functions represent symptoms of the synchronization concern. This list will be used to check the effectiveness of the mining approaches studied.

As this concern is related to 42 methods summarized in Table 1, it might be straightforward to convert each of these methods into an aspect to express all the occurrence of this method, thus generating 42 different aspects. However, this granularity of refactoring is too small because it still makes these refactored aspects code "scattered", which is inconsistent to the key objectives of AOP. It will be much more meaningful to group methods into larger aspects, each of them represents a certain mechanism or even the whole concern. Refactoring in a larger granularity can greatly improve the maintainability and comprehensibility of the refactored AOP code.

To identify these crosscutting concerns, we first apply the fan-in analysis [4] on Linux.

Fan-in analysis is based on the observation that a crosscutting concern in non-AOP systems is usually implemented by single methods in the system, which are called from numerous places in the code. The fan-in metric of a method *m* is defined as the number of distinct method bodies that can invoke *m*. (Since this metric is originally defined for the Java language, we modified it slightly

3. function-like macro is a macro "whose use looks like a function call" defined by GNU in http://gcc.gnu.org/onlinedocs/gcc-4.1.0/cpp/

Table 2. Results of fan-in analysis when mining the synchronization concern in *Linux*

Mechanisms	# Total	# Found	Coverage
atomic operation	11	5	45.45%
spin lock	11	8	72.73%
read/write spin lock	15	5	33.33%
big kernel lock	5	2	40.00%
Sum	42	20	47.61%

to support C language as discussed in Section 3.) This approach follows three steps:

Step 1. Automatically compute the fan-in metric for all the methods in the targeted source code.
Step 2. Filter the result of the first step:

- Restrict the set of methods to those having a fan-in above a certain threshold.
- Filter getters and setters from this restricted set.
- Filter utility methods such as toString().

Step 3. (*Mainly manual*) Analysis of the remaining set of methods.

The results of the fan-in analysis approach on mining synchronization concern in *Linux* are shown in Table 2. While there are totally 42 methods related to the synchronization concern as we listed in Table 1, fan-in analysis with a threshold at 10 (the threshold used in [4]) can find only 20 of them. So the coverage is 47.61%, meaning that more than half of the occurrences related to the synchronization concern cannot be found. It will definitely affect the effectiveness of refactoring if we do not find the remaining methods to completely encapsulate the concern through either manual walk-through or more comprehensive approaches.

To understand why fan-in analysis can not find other occurrences of this concern in *Linux*, we found that only 20 functions (or function-like macros) related to synchronization have a fan-in larger than (or equal to) 10 in the code (such as 156 for *spin_lock*), other functions have fan-in values smaller than 10 (such as 1 for *spin_trylock*), which will be filtered out by this approach.

To find the methods with low fan-in's, one way is to lower the threshold. However, it will bring more "fake" aspect candidates in the results, which means that more work is required to filter out these non-aspect-related methods. Note that even the threshold is set as small as 2 (which in practice will introduce too many "fake" candidates), the above mentioned function *spin_trylock* will still be left out regardlessly.

Another problem of fan-in analysis besides the selection of the threshold is that the results returned lacks information about which concern each method belongs to. In the experiment, fan-in analysis returns 116 methods as results at the threshold of 10. We have to search for the 20 methods related to synchronization and then group them into meaningful aspects manually.

If we can automatically group the methods into related groups such as those shown in Table 1, it will reduce the manual grouping effort during refactoring. Besides, if we can calculate the fan-in value for each group of methods, instead of single methods, we could identify those important but rarely called methods along with those frequently called methods together. Additionally, instead of using a fan-in threshold to cut off the results, we will use a ranking approach to recommend the clusters that are most likely to be aspects. This leads to our proposed mining approach: the clustering-based fan-in analysis.

3. Clustering Based Fan-in Analysis (CBFA)

We propose a new automated approach called Clustering-Based Fan-in Analysis (CBFA) that, through identifying aspect candidates together as groups, can improve the efficiency of aspect mining and provide better support for refactoring. CBFA adopted clustering from data mining [13] on lexical form of the source code.

Technique Overview. The process of the CBFA approach can be summarized as in Figure 1, which includes five steps:

1) The source code is first analyzed and parsed into a set of methods. In this step, we use an indexer to identify all methods in the source code and function-call relations.
2) Each of the methods analyzed is converted into a vector based on its signature.
3) An automated clustering step is then taken to group related methods together into clusters based on the similarity of vectors. The Jaccard Coefficient [13] is used in our approach.
4) In parallel with the clustering step, the fan-in value of each method is calculated based on function-call relations generated in step 1. The definition of fan-in is the same as in [4] for *Java* language systems. For *C* language systems, the definition will be extended to include function-like macros.
5) Finally, we calculate cluster fan-in of each cluster, and rank the clusters based on their fan-in values to complete the aspect recommendation step. The output of CBFA is a ranked list of method clusters, each of them is expected to be encapsulated into one (or at most a few of) meaningful aspect during refactoring.

The main steps introduced by CBFA is the adoption of lexical-based clustering in aspect mining and the recom-

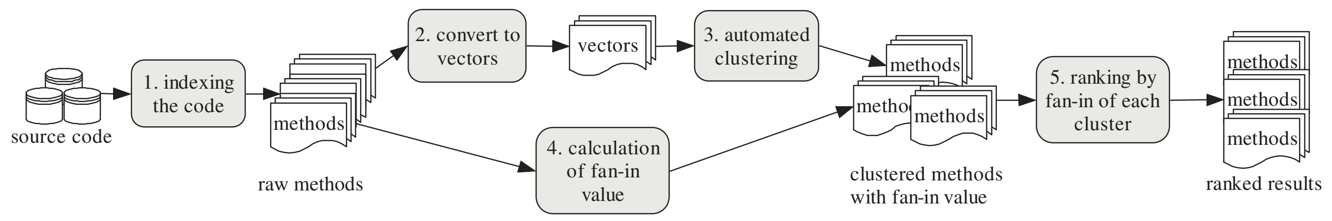

Figure 1. Overview of Our Approach

mendation step, which uses a new ranking metric.

Method Retrieving. Although many elements can be taken into account in aspect mining (such as classes, methods, annotations, and so forth), the basic element concerned in this paper is method (including function-like macro in C).

A code parser is used in the method retrieving step to extract C/C++ and Java elements in the source code, as well as the function-call relations which will be used later during fan-in value calculation.

As suggested by Marin *et. al.* [4], due to the usually high fan-in values of getters and setters in Java, which can hardly be exploited in an aspect solution, we filtered out methods with a prefix of "get" or "set" for the Java language. However, no methods need to be filtered out in systems written in C.

Vector Representation in CBFA. A vector space model (or vector model) is needed in clustering to represent each object to be clustered in aspect mining approaches. It is often used for information filtering, information retrieval, indexing and relevancy rankings [13].

Let $X = \{O_1, O_2, \ldots, O_n\}$ be the set of objects to be clustered. With a vector space model, each object is measured with respect to a set of m initial attributes $A_1, A_2, \ldots, A_m$ (usually a set of relevant characteristics of the analyzed objects) and is therefore described using an m-dimensional vector $O_i = (O_{i1}, O_{i2}, \ldots, O_{im})$, $O_{ik} \in \Re$, $1 \le i \le n, 1 \le k \le m$. Usually, the attributes associated to the objects are standardized in order to ensure an equal weight to all of them [13].

Based on the methods retrieved from the last step, it takes two steps to generate the vector representation for each method:

1) The signatures of methods are split into tokens based on naming conventions in Java and C. For example, *fireSelectionChanged* in Java is split into "fire selection changed", and *kernel_locked* in C is split into "kernel locked".
2) All these split tokens in the system are specified as m initial attributes. Each method m_i is represented as a vector $O_i = (O_{i1}, O_{i2}, \ldots, O_{im})$, where O_{in} is 1 if and only if method m_i contains the attribute A_n.

Clustering. In the clustering step, we need to specify the similarity metric and design the clustering algorithm.

Jaccard Coefficient [13] is a metric usually used in the calculation of the similarity of asymmetric binary attributes like this. It is defined as:

$$J(O_i, O_j) = \frac{N_{ij}}{N_i + N_j + N_{ij}}$$

where N_{ij} represents the total number of attributes where O_i and O_j both have a value of 1. N_i represents the total number of attributes where the attribute of O_i is 1 and the attribute of O_j is 0. N_j represents the total number of attributes where the attribute of O_i is 0 and the attribute of O_j is 1.

The similarity between two vectors $\overrightarrow{O_a}$ and $\overrightarrow{O_b}$ is thus defined as:

$$sim(\overrightarrow{O_a}, \overrightarrow{O_b}) = J(\overrightarrow{O_a}, \overrightarrow{O_b})$$

We choose Jaccard Coefficient as the similarity metric, based on the observation that in this vector model, value 0 is not as important as value 1, since methods are the "same" when they have a lot of words in common, not when they have an absence of most of the common words.

To group similar vectors into the same cluster, we use a heuristic algorithm presented in Table 3. The variable *simMin* in the algorithm represents a threshold to determine if two names are "similar" enough to be clustered. The value is chosen as 0.3 after extensive studies[4].

The input of the CBFA algorithm is a method set $M = \{m_1, m_2, ..., m_n\}$ along with vectors representing them that have been calculated in last step.

The clustering algorithm we used is pretty straightforward. At the beginning, the cluster set is empty. For each vector m_i in $M = \{m_1, m_2, ..., m_n\}$, we calculate its similarity to each other methods in existing clusters. The biggest similarity $curMaxSim$ and the index of corresponding cluster cur is recorded. If $curMaxSim$ is larger than the threshold $simMin$, this method will be clustered into the cluster with index cur, otherwise a new cluster will be created with m_i as its only method.

4. A straightforward explanation to this is that if we want to cluster two methods, whose signatures both contain two words, and one word in common, their similarity will be 1/3=0.33. Our experiments also demonstrated that 0.3 is good enough in our approach.

Table 3. Clustering Algorithm used in CBFA

Input:
the set $M = \{m_1, m_2, ..., m_n\}$ is the set of methods to be clustered, associated with vectors that represent them
$simMin(> 0)$ is the threshold, only when two methods' similarity is larger than it, should they be clustered. In CBFA, it is chosen as 0.3

Output:
$K = \{K_1, K_2, \ldots, K_p\}$, where K is a **partition** of methods set M in system, and $K_j = \{m_{j1}, m_{j2}, \ldots, m_{jl}\}$ is a cluster of similar methods

Helper functions:
$s(method_a, method_b)$: the similarity of method a and b, Jaccard Coefficient is used in our approach.

Algorithm CBFA:

```
$k \leftarrow \Phi$
$K \leftarrow \Phi$
for each $m_i \in M$ do
  $cur \leftarrow 0$
  $curMaxSim \leftarrow 0$
  for each $k_j \in K$
    for each $m_l \in k_j$
      if $s(m_i, m_l) > curMaxSim$
        $curMaxSim \leftarrow s(m_i, m_l)$
        $cur \leftarrow j$
  if $curMaxSim > simMin$
    $k_{cur} \leftarrow k_{cur} \bigcup \{m_i\}$
  else
    $k \leftarrow \{m_i\}$
    $K \leftarrow K \bigcup \{k\}$
return $K$
```

Fan-in Value Calculation. In parallel with the clustering step, we calculate the fan-in value of each method.

As mentioned, the calculation of fan-in values is performed using the definition in [4] for *Java*. However, for *C* language system, we notice that this approach should be modified as follows:

1) Because there is no polymorphism in *C*-based systems, the *fan-in* is just the number of times a specific method was called throughout the source code.
2) Besides functions, there are also function-like macros acting like functions in *C*. These function-like macros should be treated as functions while calculating fan-in values.

The calculation is straightforward based on the function-call relations generated in step 1.

Ranking Metrics. Numerous clusters will be returned after the clustering step in a large system, thus a good ranking algorithm is necessary to rank the resulting clusters in order to provide proper aspect recommendation.

Zhang *et al.* [9] used a random walk model motivated by the page-rank algorithm [16]. The idea underlying the page-rank algorithm is to generate ranks reflecting the degrees of "popularity" and "significance". As a much simpler metric, fan-in values also represent "popularity" and "significance" of methods since a method called more frequently is typically more popular and thus more significant.

After the clustering step, the fan-in of a whole cluster is more useful than that of a single method, because we expect that the whole cluster together, instead of a single method in it, should be refactored into an aspect. Therefore we define a new metric called cluster fan-in as below.

The *cluster fan-in* F of a cluster C is defined as:

$$F(C) = \sum f(m_i), m_i \in C$$

where function f returns the fan-in value of a single method.

The final results of CBFA are ranked based on the *cluster fan-in* values of all the clusters in a descending order to recommend the clusters based on their significance.

4. Evaluation

4.1. Experimental Setups

We implement CBFA as a plug-in on Eclipse[5]. For *C* language systems, we use C/C++ Develop Tools (CDT) to index the source code, and calculate the fan-in value of each method. The version of CDT we used is 4.0.1. For *Java* language systems, we use Java Develop Tools (JDT) to perform the indexing and fan-in calculation tasks. The version of JDT we used is 3.3.1.

We will analyze two systems in this paper. As a popular open source operating system, *Linux* is a typical *C* language legacy system. The version we analyzed is 2.4.18. Due to the code size limit of CDT, we can not analyze the whole *Linux* system. Instead, we analyzed a subsystem of *Linux* without net, file system, and platform (except i386) related code. The subsystem we analyzed consists of 1064 ".c" files and approximately 84K lines of code.

The *JHotDraw* framework is a *Java* GUI framework for technical and structured graphics. Since it is designed as an exercise to show how to use design patterns, it was analyzed by many mining research groups to evaluate their mining approaches [4], [10], [12]. The version we analyzed is 5.4b, which includes about 12K lines of code.

To evaluate the mining efficiency of CBFA, we first compare CBFA on several well-known CCCs in the literature (e.g., undo and persistence) with three state-of-the-art aspect mining approaches in a top-down evaluation. The techniques evaluated include fan-in analysis [4], identifier analysis [8], and a dynamic approach [7]. We then explore the capability of CBFA to find other less known aspects within a large system using a bottom-up approach.

5. www.eclipse.org

4.2. Metrics

Two metrics are used in our evaluation: *concern coverage* and *true positives* [6]. They are based on precision/recall measures used in information retrieval systems. We formally define them as follows.

First, we define the set $Correct(c, tech)$ and $Wrong(c, tech)$ for a certain mining technique $tech$ as follows:

$$Correct(c, tech) = Total_c \bigcap Cand_{tech}$$

$$Wrong(c, tech) = Cand_{tech} - Total_{ccc} \bigcap Cand_{tech}$$

where $Total_c$ stands for the method set related to a concern c; $Total_{ccc}$ is the set includes all methods that are considered aspect related. $Cand_{tech}$ is produced by a mining technique $tech$ as the set of aspect candidates for concern c.

Using the above definitions, we can calculate the concern coverage and true positives as follows:

$$ConcernCoverage(c, tech) = \frac{|Correct(c, tech)|}{|Total_c|}$$

$$TruePositives(c, tech) = 1 - \frac{|Wrong(c, tech)|}{|Cand_{tech}|}$$

As most of the existing aspect mining techniques do not group the results based on concerns, $Correct(c, tech)$ measure is defined using the total set of crosscutting concerns from the system. In such cases, only an average true positives can be provided. It will be a little difficult to calculate $Wrong(c, tech)$ because it is subjective to decide whether a method is related to a certain concern.

Typically, getting high concern coverage is more important than getting high true positives. After all, assuming that our ultimate goal is to encapsulate all methods related to a specific concern into an aspect, then as long as you get less than 100% coverage, you have to go through the rest of code (which may be millions of lines of code) to find the remaining ones. On the other hand, if you do not get 100% true positives, you only have to go through the mining results to eliminate the "fake" ones, which in most cases will be much smaller than the entire code base.

4.3. Top-Down Approach

In order to evaluate the performance of CBFA, we first choose several well-known CCCs in JHotDraw and Linux, and compare the mining efficiency of CBFA with three state-of-the-art mining approaches in a top-down approach.

6. This name comes from "false positives" used in many recent work [4], [10], [12].

Concerns Analyzed. We choose five widely analyzed CCCs [4], [10], [12] in JHotDraw, which include:

1) *Undo*: an undo concern in a graphical editor like JHotDraw performs undo and redo activities. It is an implementation of the Command design pattern [17]. A more detailed discussion about this concern and how to refactor it can be found in [4].
2) *Observer*: this concern implements the Observer design pattern. As there are at least eight different implementations of this concern in JHotDraw, we choose figure selection observer as an example.
3) *Iterator*: Iterator is a widely used design pattern in Java. It is commonly used to traverse the collections in Java. Two other implementations, which are used to iterate figures and handles, can be found in JHotDraw.
4) *Visitor*: the Visitor design pattern is also widely used in Java to visit objects.
5) *Persistence*: persistence is a concern in JHotDraw to save and reload items.

As it is hard to identify all methods that are related to a certain concern (extremely difficult for complex concerns such as Observer), in our evaluation, a set of methods are considered to be an indication of a concern based on three criteria below:

1) they should be relevant to this concern,
2) scattered in the system, and
3) the programmers should be able to locate *all* the classes related to that concern based on these methods.

A detailed list of the methods we identified based on the above criteria can be found in Table 4.

For Linux, because there are few well-known CCCs, we will use the synchronization concern describe earlier as the test case, which can be found in Table 1.

Techniques Compared. Three well-known mining techniques are taken into comparison, which include fan-in analysis [4], identifier analysis [8] and dynamic analysis [7].

Fan-in analysis has been introduced in Section 2. The public available tool *FINT*[7] is used during our evaluation. A fan-in threshold is required during the evaluation, we choose 10 for JHotDraw as suggested by Roy *et. al.* [12].

Identifier analysis performs Formal Concept Analysis (FCA) [18] on generated elements, such as method names. The results of this approach are called "concepts", which include methods whose names share a certain word. For example, "readInt" and "readDouble" are both included in the "read" concept. A threshold of concept size is required during analysis: it is chosen as 10, also as suggested by

7. Available from http://swerl.tudelft.nl/bin/view/AMR/FINT

Table 4. List of CCCs analyzed in *JHotDraw*

CCC	Related methods
Undo(7)	*undo, redo, execute, pushUndo, popUndo, pushRedo, popRedo*
Observer (5)	*addFigureChangeListener, removeFigureChangeListener, willChange, changed, figureChanged*
Iterator(6)	*next, hasNext, nextFigure, hasNextFigure, nextHandle, hasNextHandle*
Visitor(4)	*visit, visitFigure, visitHandle, visitFigureChangeListener*
Persistence (16)	*read, readStorable, readString, readInt, readLong, readColor, readDouble, readBoolean, write, writeStorable, writeString, writeInt, writeLong, writeColor, writeDouble, writeBoolean*

Roy *et. al.* [12]. Since the lack of public available tools, we implemented a prototype tool, which performs FCA on the method names.

Dynamic analysis collects the trace information while executing certain use cases. The *Dynamo*[8] aspect mining tool is used to evaluate the dynamic analysis approach, the use cases adopted in [10], [12] are applied in the evaluation.

However, none of the existing tools (*FINT* and *Dynamo*) is applicable for Linux. We have implemented a new fan-in analysis tool that is able to work on C code [11]. We also extended the prototype tool implemented for identifier analysis in Java to make it applicable for C. Dynamic analysis approach is not evaluated on Linux due to the lack of use cases.

Result Analysis. The performance of each technique is shown in Table 5 and Table 6. For mining approaches that provide meaningful groups of methods, an extra number bracketed in concern coverage indicates how many groups are provided for a certain concern. An ideal mining approach should provide exactly one group (normally, less than a few groups is also acceptable) for each concern.

From the results for JHotDraw from Table 5, we can see that CBFA can achieve a considerable higher concern coverage for the concerns compared to other techniques, while encapsulating each concern in only one or two clusters. CBFA is also the best while comparing the true positives metric.

Fan-in analysis tends to miss methods with a low fan-in value. As shown in section 2, it misses methods whose fan-in values are smaller than the threshold. For example, the visitor concern contains six methods, whose fan-in values are all smaller than the threshold during evaluation. Thus, fan-in analysis can find none of these methods.

Identifier analysis returns 85 concepts at the chosen threshold (10) in JHotDraw. However, it tends to omit smaller groups. For example, the concept containing the iterator concern has only six methods, thus it is filtered out by the chosen threshold [9]. The case with the visitor concern is similar.

Another benefit of CBFA over identifier analysis is that, methods that can not be included in the same concept in the latter may be clustered together by CBFA if they are similar enough. For example, CBFA can find 6 methods (except for *execute*) in exactly one cluster for the *undo* concern, however it requires at least two concepts(concept "undo" and "redo") for FCA to cover them. In CBFA, *undo* and *popRedo*, which can never be included in the same concept in FCA, can be clustered together with the help of *popUndo*, which is similar to both of them.

Dynamic analysis tends to miss methods which are not called in the use cases. For example, iterator concern is completely omitted because methods related are not called in the use cases. Even when a certain concern is included in the use cases, the concern coverage may be still low (for example, since the persistence use case only contains write operations, and the read operations are not covered by it, the coverage of persistence concern is still low using this use case).

The performance of fan-in analysis in Linux is similar to JHotDraw. Identifier analysis generates 205 concepts in Linux. As the concepts of selected concerns in identifier analysis are large enough (even the concept containing the *big kernel lock* concern has a size larger than 10, because it contains other methods that not belonging to this concern), we can see from the results that it performs as well as CBFA in these cases.

4.4. Bottom-Up Approach

We analyze the capability of CBFA to find other less known aspects within both JHotDraw and Linux in a bottom-up analysis.

From the top recommendations of CBFA, we take the first ten CCCs and compare the performance of CBFA with the other three techniques discussed in section 4 on these CCCs. As concern coverage is more important than true positives, we only evaluated concern coverage for each system due to the space limitation.

Since there is little published knowledge about most of these identified concerns, we looked at the source code and documentation of applications in order to achieve a clear picture of the application to perform a manual analysis of the results.

8. Available from http://star.itc.it/dynamo/ under GPL

9. We have noticed that the current approaches of identifier analysis could also be extended in order to provide a ranked list of concepts, instead of using a fixed threshold. A discussion of the recommendation quality of this extended version and CBFA is given in Section 5.1.

Table 5. Comparison of CBFA and Other Techniques for JHotDraw

Concern	#methods	Concern Coverage				True Positives			
		CBFA	Fan	Iden	Dyn	CBFA	Fan	Iden	Dyn
Undo	7	85.71%(1)	42.86%	85.71%(2)	57.14%(1)	100%	N/A	50%	63.89%
Observer	5	80%(1)	100%	60%(1)	40%(1)	86.49%	N/A	73.08%	62.5%
Iterator	6	100%(1)	83.33%	0%(0)	0%(0)	100%	N/A	N/A	N/A
Visitor	4	100%(1)	0%	0%(0)	75%(1)	86.49%	N/A	N/A	50%
Persistence	16	100%(2)	37.5%	100%(2)	43.75%(1)	80%	N/A	75%	70%
AVERAGE	N/A	93.14%	52.87%	49.14%	43.18%	90.60%	73.88%	66.03%	61.60%

Table 6. Comparison of CBFA and Other Techniques for Linux

Concern	#methods	Concern Coverage			True Positives		
		CBFA	Fan	Iden	CBFA	Fan	Iden
atomic operation	11	100%(1)	45.45%	100%(1)	91.67%	N/A	90.91%
spin lock	11	100%(1)	72.73%	100%(1)	87.84%	N/A	100%
read/write spin lock	15	86.67%(2)	33.33%	86.67%(2)	80.95%	N/A	75%
big kernel lock	5	100%(1)	40%	100%(1)	87.84%	N/A	75%
AVERAGE	N/A	96.67%	47.88%	96.67%	87.08%	52.35%	85.23%

Table 7. Bottom-up Approach Results for JHotDraw

Concern	CBFA	Fan	Iden	Dyn
composition(1)	100%	100%	100%	0%
mouse handler(15)	86.67%	26.67%	100%	28.57%
zoom(6)	100%	0%	0%	0%
factory method(21)	100%	2.38%	100%	2.38%
iterator(6)	100%	83.33%	0%	0%
persistance(16)	100%	37.5%	100%	43.75%
undo(7)	85.71%	42.86%	85.71%	57.14%
manage handles(16)	75%	0%	50%	100%
observer(5)	80%	100%	60%	40%
draw(25)	92%	12%	96%	4%
AVERAGE	91.94%	40.47%	69.71%	27.58%

Table 8. Bottom-up Approach Results for Linux

Concern	CBFA	Fan	Iden
FPU(72)	90.27%	16.67%	100%
pgd(7)	100%	14.29%	100%
pte(14)	92.86%	7.14%	100%
pmd(9)	100%	0%	100%
spin lock(11)	100%	72.73%	100%
read/write spin lock(15)	86.67%	33.33%	86.67%
big kernel lock(5)	100%	40%	100%
ipc(7)	100%	57.14%	0%
printk(1)	100%	100%	0%
local irq(5)	80%	80%	0%
AVERAGE	94.98%	42.13%	68.67%

Result Analysis. The first ten CCCs recommended by CBFA in JHotDraw are listed in Table 7. The number shown in the bracket indicates the number of methods related to that concern. The four CCCs whose names are bolded are the concerns that have been analyzed before. The concern called “manage handles” is analyzed in [10]. The intentions of other concerns manifest themselves in their names quite clearly, so we skip detailed introduction to each concern.

One observation from the results is that CBFA can identify new concerns that none of current approaches can find (such as the *zoom* concern). The fan-in analysis can not find the *manage handles* concern, and it only get a concern coverage less than 20% for the *factory method* and *draw* concerns. The identifier analysis omits two concerns (*zoom* and *iterator*) completely. Dynamic analysis performs the worst among these approaches.

The results for Linux are shown in Table 8, except for dynamic analysis. The fan-in analysis approach missed the *pmd* concern, and only get a concern coverage less than 20% for the *FPU*, *pgd* and *pte* concerns. The identifier analysis misses three concerns because of the the small sizes of the concepts containing them (for example, the size of the concept that contains *printk* is only 2). While the performance of CBFA is similar as identifier analysis in the previous top-down approach evaluation, it surpasses the latter in these cases.

Based on the results obtained in both top-down and bottom-up approaches, CBFA can provide accurate recommendations while improving aspect mining coverage significantly compared to other state-of-the-art mining approaches.

5. Discussions

5.1. Recommendation Quality

As CBFA provides automated recommendation of aspect candidates in the result, the recommendation quality plays a very important role in the evaluation. Here we use the number of cluster candidates a user need to examine before reaching the targeted aspect as a quality metric, since the fewer recommendations the user needs to examine, the more efficient an mining approach is.

We use the five CCCs in JHotDraw as discussed in section 4, to evaluate the recommendation quality of CBFA. As these concerns are widely analyzed in the literature [4], [10], [12], we consider that they are somewhat "representative".

As fan-in analysis do not group the results, and dynamic analysis's recommendation quality strongly depends on the manually chosen use cases, only the recommendation quality of identifier analysis is compared with CBFA. Although the latter does not provide a ranked results, as it filters out the small sized clusters, we apply the cluster size as its ranking metric.

During our evaluation, all of these five concerns can be covered within the top 42 clusters that was recommended by CBFA, while the extended identifier analysis requires the user to examine at least 151 concepts to find all these five concerns, which causes a considerable larger manual effort to identify the concerns needed.

In short, the recommendation quality surpasses that of extended identifier analysis: the quality of the former was roughly 1 in 8 (i.e., 5 valid CCCs in the 42 identified clusters), compared to 1 in 30 for the latter.

5.2. Using CBFA to Support Refactoring

This paper is focused on how to automatically identify aspect candidates efficiently. However, this is only the first step towards aspect-oriented rewrite of existing legacy system. These candidates must still be converted into aspect-oriented code using certain refactoring mechanism. Here we discuss how CBFA can efficiently support the automation of refactoring process, with help of the some related refactoring work [19], [20].

Usually, the refactor tools require a marked source code, where all the identified locations related to possible concerns are marked. Based on the identified clusters in the system by CBFA, This can be accomplished by identifying all occurrences of all methods in the clusters, with help of a search module provided by most IDEs, such as the Eclipse/CDT we adopted.

Next, these marked potential aspects should be extracted into pointcuts and advices. The human-guided automated technique proposed by Binkley et. al. [20] can be adopted to extract these code. One benefit of using CBFA as the aspect mining approach is that, as the aspect candidates provided by CBFA are grouped into meaningful clusters as shown in our evaluation, these generated *pointcuts* and *advices* code can usually be encapsulated in the (almost) correct aspects, and thus reduce the human inference in this step.

Finally, we can rewrite the raw *pointcuts* by an automated inference of pointcuts in aspect-oriented refactoring [19]. For example, join points in *changeSelection()* pointcut of *afterDrawApplication* aspect [19] in JHotDraw below:

```
public void DrawApplication.clearSelection()
public void DrawApplication.toggleSelection(Figure
```

could be rewritten as one pointcut expression:

```
public void DrawApplication.*Selection()
```

This shows that, with the help of several existing work, CBFA can be adopted to accomplish the whole process of refactoring legacy software in aspect-oriented systems automatically or at least semi-automatically.

6. Related Work

Aspect mining is an important step in aspect refactoring. The goal of aspect mining is to identify aspect candidates in the source code automatically or semi-automatically. The techniques related to our approach can be classified into the following categories.

A lot of aspect mining efforts are lexical based. Tourwe and Mens [8] perform FCA on generated elements, which can be any source code artifacts. In their initial work, however, they only considered classes, methods and formal parameters. Shepherd *et al.* [5] used *Lexical Chaining* for aspect mining. The main difference between CBFA and these two approach is that: we used a clustering algorithm commonly used in information retrieving to group the elements, instead of FCA used in [8] and lexical chaining used in [5]. The inherent limitation of all lexical based approaches (including CBFA) is that, they are not capable of finding methods that do not follow a good naming convention.

Another natural language processing based mining approach [21] uses a search-based approach with queries performed over a program model that captures the *action-oriented relations between identifiers* in a program. This technique is capable of finding action-oriented concerns, such as *playTrack* and *playCD*, while CBFA can find other concerns as well as action-oriented concerns.

Fan-in analysis [4] are motivated by the symptom that, CCCs are usually implemented as methods in the system, which is called from numerous places in the code. Another example of an implementation idiom of CCCs is "code duplication". Bruntink *et al.* [3] use clone detection to find aspect candidates in a C-based system *ASML*. The main limitation of such approaches is that, they only provide individual aspect candidates, without grouping similar elements to groups. This directly makes an unfavorable impact on the concern coverage and requires more manual efforts to group them into meaningful aspects.

Besides static approaches mentioned above, dynamic analysis approaches, such as DynAMiT [2] and Dynamo [7], are proposed. The limitation of such approaches is that their performances rely heavily on the use cases to be executed. However, they could be used as complementary techniques with static ones to identify additional concerns.

7. Conclusion

In this paper, we propose an automated aspect mining approach called Clustering-Based Fan-in Analysis (CBFA). CBFA adopts a lexical based clustering approach to group methods related to the same crosscutting concern together, and automatically recommend aspects based on the *cluster fan-in* ranking metric. Experiments in *Linux* and *JHotDraw* shows that not only can CBFA provide accurate recommendations, it can also improve aspect mining coverage significantly compared to other state-of-the-art mining approaches.

Future work of CBFA include evaluating CBFA on more systems, refining its algorithms in order to achieve better performance, as well as a more in-depth research on how to automatically generate aspect-oriented code based on the mining results of CBFA.

Acknowledgement

We are grateful to Prof. Hong Mei, for his guidance and support in both the research and paper writing process. We want to thank Dr. Lu Zhang for discussions and comments on the earlier draft of this paper. We would also thank the anonymous reviewers for their insightful comments. The authors are sponsored by the National High Technology Research and Development Program(863) Program of China under Grant No. 2007AA01Z462, 2007AA010304 and 2008AA01Z133.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *proc. of European Conference on Object-Oriented Programming(ECOOP)*, June 1997.

[2] S. Breu and J. Krinke, "Aspect mining using event traces," in *proc. of International Conference on Automated Software Engineering (ASE)*, 2004, pp. 310–315.

[3] M. Bruntink, A. v. Deursen, T. Tourwe, and R. van Engelen, "An evaluation of clone detection techniques for identifying crosscutting concerns," in *Proc. of International Conference on Software Maintaince (ICSM)*, 2004, pp. 200–209.

[4] M. Marin, A. V. Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 1, pp. 1–37, 2007.

[5] D. Shepherd, T. Tourwe, and L. Pollock, "Using language clues to discover crosscutting concerns," in *Workshop on the Modeling and Analysis of Concerns*, 2005.

[6] D. Shepherd, L. Pollock, and K. Vijay-Shanker, "Towards supporting on-demand virtual remodularization using program graphs," in *proc. of the International Conference on Aspect-oriented Software Development (AOSD)*, 2006, pp. 3–14.

[7] P. Tonella and M. Ceccato, "Aspect mining through the formal concept analysis of execution traces," in *proc. of 11th Working Conference on Reverse Engineering (WCRE)*, 8-12 Nov. 2004, pp. 112–121.

[8] T. Tourwe and K. Mens, "Mining aspectual views using formal concept analysis," in *proc. of Source Code Analysis and Manipulation Workshop (SCAM)*, 2004, pp. 97–106.

[9] C. Zhang and H. A. Jacobsen, "Efficiently mining crosscutting concerns through random walks," in *proc. of the International Conference on Aspect-oriented Software Development (AOSD)*, 2007, pp. 226–238.

[10] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe, "A qualitative comparison of three aspect mining techniques," in *proc. of the 13th International Workshop on Program Comprehension*, 2005, pp. 13–22.

[11] D. Zhang, Y. Guo, Y. Wang, and X. Chen, "Toward efficient aspect mining for linux," in *Asia-Pacific Software Engineering Conference (APSEC)*, 4-7 Dec. 2007, pp. 191–198.

[12] C. K. Roy, M. G. Uddin, B. Roy, and T. R. Dean, "Evaluating aspect mining techniques: A case study," in *proc. of 15th IEEE International Conference on Program Comprehension (ICPC)*, 26-29 June 2007, pp. 167–176.

[13] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[14] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schrder-Preikschat, "An aspect-oriented implementation of interrupt synchronization in the PURE operating system family," in *proc. of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, June 2002.

[15] R. Love, *Linux Kernel Development*. Novell Press, 2005.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," *Technical Report, Stanford Digital Library Technologies Project*, 1998.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[18] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*. Spring-Verlag, 1999.

[19] P. Anbalagan and T. Xie, "Automated inference of pointcuts in aspect-oriented refactoring," in *proc. of the International Conference on Software Engineering (ICSE)*, 2007, pp. 127–136.

[20] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, "Automated refactoring of object oriented code into aspects," in *proc. of International Conference on Software Maintenance (ICSM)*, 2005, pp. 27–36.

[21] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *proc. of the International Conference on Aspect-oriented Software Development (AOSD)*, 2007, pp. 212–224.