Mining Crosscutting Concerns through Random Walks

Charles Zhang, *Member, IEEE,* and Hans-Arno Jacobsen, *Senior Member, IEEE*

***Abstract*—Inspired by our past manual aspect mining experiences, this paper describes a probabilistic random walk model to approximate the process of discovering crosscutting concerns in the absence of the domain knowledge about the investigated application. The random walks are performed on the concept graphs extracted from the program sources to calculate metrics of "utilization" and "aggregation" for each of the program elements. We rank all the program elements based on these metrics and use a threshold to produce a set of candidates that represent crosscutting concerns(CCs). We implemented the algorithm as the** Prism CC miner (PCM) **and evaluated** PCM **on Java applications ranging from a small-scale drawing application to a medium-sized middleware application and to a large-scale enterprise application server. Our quantification shows that** PCM **is able to produce comparable results (95% accuracy for top 125 candidates) with respect to the manual mining effort.** PCM **is also significantly more effective as compared to the conventional approach.**

I. INTRODUCTION

Aspect mining, or more precisely, the mining of crosscutting concerns [1](CCs), refers to the activity of locating program elements in source code that pertain to non-modularized coding concerns. For large software systems consisting of millions of lines of code, automatically revealing non-localized concerns greatly benefits many software engineering tasks such as program comprehension and maintenance [2], software customization [3], [4], and the aspect-oriented design of new applications. This need has given rise to active research on aspect mining for close to a decade [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15].

A. Research context

The aspect mining approach presented in this paper belongs to a category of mining techniques [7], [8], [13], [15] that statically analyze the source code structure of the target software system. These techniques are widely applicable as they do not require additional information such as the revision history [11], [14] or runtime information [9], [10], [14], which are not always available or easily obtained.

Directly finding CCs with high precision using only the program sources is challenging as it requires program analysis tools to reason about application semantics such as "programmer intention" or "functional orthogonality". Consequently, a large majority of the existing static approaches [7], [8], [13] approximate CCs using code-level techniques such as *fan-in analysis* and *clone detection*. Phenomena such as *cloning* or *high degree of fan-in* signify the characteristic of "concern scattering", i.e., the *frequent* and *non-localized* impact of crosscutting concerns on the program sources.

B. Contributions

Our main observation is that the fan-in and the cloning-based CC mining approaches essentially favor program elements that appear frequently, which can incur a large number of false positives and erroneous conclusions. What conventional methods ignore is that the modular structure of the whole program can be viewed as a network of concern relationships, which can be leveraged to improve the mining efficiency. Let us illustrate this observation through an example.

Our example[1] is an adapted version of the "telecom" example from the source distribution of the AspectJ compiler, where it serves as an illustration of aspect-oriented programming. We re-implemented the features "Billing" and "Timing" in plain Java, which were originally implemented as aspects. We converted the original "print" statements to calls to a `Logger`, which exclusively uses the class `StorableOutput` to write persistent logs. We added the persistence capabilities into classes `Call`, `Connection`, and `Customer` by having them implement the `Storable` interface. This represents the usage homogeneity in type definitions. Therefore, the CC candidates in our example should be the following: `Logger, Storable, StorableOutput, Timer`, and `Billing`.

Figure 1(A) presents the UML diagram of the telecom application. Figure 1(B) shows the relationship graph of all the class types after superimposing the type relations onto the call graph. A count of the fan-in degrees for all the classes produces the following ordered set with descending degrees: `Customer`(5), `Logger`(4), `Connection`(4), `Storable`(3), `Timer`(1), `Billing`(1), `Call`(1), `Local`(1), `LongDistance`(1), and `StorableOutput`(1). Taking the top ranked elements, i.e., `Customer, Logger, Connection, Storable`, we have not only mistakenly included two classes, `Customer` and `Connection`, representing the core functionality of the application, we have also missed three other types: `StorableOutput, Timer`, and `Billing` as they have low fan-in degrees.

As illustrated by the example, the error made by the fan-in based method, or by other frequency-based approaches including the use of clones, is because they fail to observe two important facts about the whole modular structure of the program. First, the relation between any two modules is *bi-directional*. There is also the fan-out relation in addition to

[1]This example is publicly available at http://www.cse.ust.hk/~charlesz/pcm/tele.zip

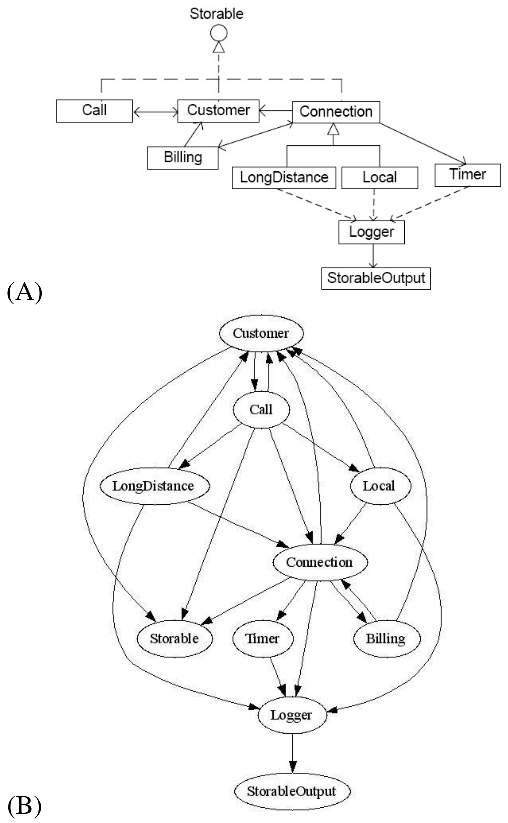


Fig. 1. A: UML class diagram. B: Concept graph

the fan-in relation. In our example, if we also consider the fan-out relation of the type `Customer`, then its transitive closure along the out-going edges includes every element of the graph. This is a strong clue that the type `Customer` is a key component of the program, i.e., a part of the primary decomposition of the program, because its functionality depends on most of the program features. Second, the module relation is also *transitive*. If we inspect beyond the immediate neighbors, we can conclude that `StorabelOutput` belongs to the same CC as `Logger` does because the former is exclusively used by the latter. Therefore, by reasoning over the complete graph structure and taking into account both fan-in and fan-out relations as well as their transitive nature, we can avoid many false classifications made by counting fan-in frequencies alone.

From these observations, we propose a novel CC mining algorithm that more accurately locates crosscutting concerns by making use of the whole modular structure of the program. We model the manual process of CC investigation as probabilistic random walks over the entire program sources, capturing both the bi-directional and the transitive characteristics of module relations. The walks are performed on the concept graph extracted from the program sources. The nodes of the graph represent program elements such as components, packages, classes, methods, or arbitrary collections of program elements subject to custom definitions. The directed edges represent the coupling relations between program elements. The random walks are carried out along both the forward and the reverse directions of the concept graph edges. Consequently, we obtain two metrics for each node, one representing its *degree of utilization*, i.e., how much the implementation of other elements is dependent on this node, and the other representing its *degree of aggregation*, i.e., how much of its implementation is dependent on other nodes. Our algorithm treats a high degree of utilization of a program node as a positive indicator for it being a CC concern. Simultaneously, we use the degree of functional aggregation as a negative indicator for the intuitive reason that a high degree of aggregation signifies a major program functionality, a characteristic of the so-called "core" functionality of the program. The output of our algorithm is a set of ranked components based on the two metrics: *utilization* and *aggregation*.

The actual values of utilization and aggregation are computed by a specific random walk algorithm. Our previous work [16] studied the use of the PageRank [17] algorithm. In this paper, we have generalized our approach to transparently work with both the PageRank algorithm and another well-known algorithm, HITS [18]. This generalization requires the adaptation of the non-probabilistic nature of the HITS algorithm to the stochastic nature of our random walk model. It also requires the redefinition of how our approach ranks program elements based on the utilization and aggregation values computed by either the PageRank or the HITS algorithms. In our generalized model, the PageRank and the HITS algorithms are interpreted as two alternative probabilistic judgment processes for calculating utilization and aggregation values. The analytical differences between these two random walk algorithms are of a mathematical nature and beyond the scope of our contribution. We empirically show that, in many cases that we evaluated, the HITS-based probabilistic judgement produces better results compared to our earlier work, which is based on the PageRank algorithm. However, PageRank-based judgement produced superior results in some of the experiments. The choice of which algorithm to use can be specified as a parameter to our random walk model, which facilitates the final judgements by the users of PCM.

We have implemented our approach as the Prism CC Miner (PCM), an Eclipse plug-in that seamlessly provides the CC mining capability to Java programmers via the Eclipse IDE. Although PCM operates fully autonomous, we have also developed and implemented a declarative language, called Prism Query Language (PQL), to allow for the customization of the mining process by injecting knowledge about the target application into it. Such customizations can improve the quality of the mining results significantly. To quantify the quality of PCM, we first compare the mining results of PCM to our earlier manual mining efforts [19], [6]. Our experiments show that, on a medium-sized middleware application, PCM is capable of producing effective results (i.e., a 95% precision for the Top-125 elements and a 73% recall over the entire oracle data set), as opposed to months of manual mining effort. Compared to the related approaches, PCM is at least 2 X more effective in finding crosscutting concerns, using our middleware data set and the jhotdraw data set. In each of these evaluations, we empirically analyzed the performance differences of the effects of the PageRank and HITS methods in our approach. We also conducted a case study, in which we confirmed by consulting with AspectJ developers the discovery of new crosscutting concerns via PCM.

We make the following contributions in this paper:

1. We first describe the concept graph and the random walk model for computing utilization and aggregation values of elements in the concept graphs. This model emulates and precisely quantifies a manual mining process.

2. We present the detailed design of the PCM CC mining algorithm, including the construction of the concept graph, the natural and differentiated ranking methods, and the idea of domain knowledge injection.

3. We provide a thorough evaluation of the properties of PCM. The evaluation includes the quantification through comparing to our previous manual mining results as well as the qualitative assessments by consulting with domain experts. We also provide a comparative study to a related mining approach based on fan-in analysis [7]. In addition, we discuss the differences between the results of our tool and the manual judgement as well as the limitations of the proposed algorithm.

The rest of the paper is organized as follows: Section 2 describes our algorithm in detail; Section 3 presents the implementation of the algorithm; the evaluation of the algorithm is described in Section 4, including both quantitative and qualitative measures; Section 5 introduces the current aspect mining approaches and sets our work apart from related approaches.

II. The Prism CC Mining Algorithm

The goal of CC mining is to find program features that have non-localized footprints in the source code. However, being non-localized is only a necessary but not sufficient condition for the high precision classification. For instance, through a simple tally in the source code of the database engine, Derby, we found that the class type `Table` is among the most frequently used data structures. It is obvious that `Table` is a centerpiece of the database engine architecture and frequently used due to necessity. Our first insight is that, as a form of duality, this non-localized presence systematically increases not only the usage frequency of CC features but also the degree of the functional aggregation of the core components that use these CC features. Consequently, if we consider not only how widely a feature is utilized but also how much it depends on other program features, we acquire an additional perspective of a different dimension that helps us to improving the quality of mining results. Our second insight is that the effect of aggregation and utilization is transitive along the directions of the module relation network as illustrated by the example in Section I. Due to the transitivity, the assessment of these effects need to be conducted accumulatively considering the whole modular structure of the program.

Our algorithm exploits this phenomenon and uses the *accumulated* usage frequency, which is more accurate than the "fan-in" metric, as a positive indicator of the likelihood for a feature to be classified as CC. More importantly, our algorithm also uses the degree of the *accumulated* functional aggregation as a negative indicator. Intuitively speaking, of two features that have similar "fan-in" degrees, the one having higher "fan-out" complexity is less likely to be a CC feature as compared to the other. The primary task of our algorithm is to assign numeric values to these two measures and to calculate the likelihood of representing CCs for each of the source elements in the program under investigation. In the rest of the section, we first outline a random walk model used for the calculation of utilization and aggregation, followed by an analytical description. We then present the CC mining algorithm in detail. The implementation of this algorithm is presented in detail in Section III.

A. CC mining as random walks

Assuming no domain knowledge with respect to the program semantics, a programmer investigates crosscutting concerns in a probabilistic manner, which we model as a random walk process. The target of the investigation is the program source comprised of a set of modules. And the investigator examines the pairwise relationships between two modules. If a module A uses the module B as part of its implementation, we say module A "aggregates" the function of module B and module B is "utilized" by module A. As explained earlier, these "aggregation" and "utilization" properties are transitive in nature. We also state a *fairness* precondition that, before this process starts, every program module has an equal chance of representing either a core or a CC feature. In Figure 2, we illustrate the aggregation and utilization relationships among seven modules, of which the arrow represents the "use" relationship such as method calls.

The random walk is carried out when the programmer randomly pick, with equal chances, a module to start her investigation. To follow the duality intuition, the investigator updates two counters for every investigated module, one for utilization and the other for aggregation. For the currently investigated module, m, the investigator first needs to decide what the next module is to inspect. The investigator can follow the aggregation direction, pick any module, m', that uses m, and updates the aggregation counter of m'. Or she can follow the utilization direction, pick any module m' used by m, and updates the utilization counter of m'. This decision of which direction to follow gives rise to two kinds of judgement philosophy for the determination of crosscutting concerns, which we explain as follows.

a) Reinforcement: The *reinforcement* judgement represents a *mutual reinforcing belief* that, if the current module is more likely to be a CC element, i.e., having its utilization counter incremented, the module that use this module is also more likely to be a core element, i.e., having its aggregation counter incremented. And vice versa. Therefore, the reinforcement random walk updates the utilization and the aggregation counts back and forth with a single random walk. That is, for each module m_b, we first increment the aggregation count of all modules that use m_b. We then hop backwards to one of these modules, e.g., m_a. At m_a, we update the utilization counters along the *utilization* direction for every module that m_a uses, and hop forward to one of these modules. We then repeat the same cycle.

b) Propagation: The *propagation* judgement represents the *transitive belief* that, if the current module is more likely to be a CC element, i.e., having its utilization counter incremented, the module that the current module uses is also likely to be a CC element, i.e., having its utilization counter incremented. The propagation random walk updates the utilization and the aggregation counts through two different walks. In the *utilization walk*, we first randomly pick a module m_a and increments its utilization count. We then simply follow the utilization direction and randomly pick the next module to repeat the same procedure. In the *aggregation walk*, we perform the identical procedure except that we only follow the aggregation direction and update the aggregation counter.

Following either the reinforcement or the propagation judgement, the two counters record the likelihood for the investigator to visit the corresponding modules when inspecting the utilization and the aggregation properties after a large number of random steps. More frequent visits to a particular module signify its prominence compared to other peer modules in the program in terms of the investigated property. In our manual CC mining experience, the frequent inspection of a module strengthens the programmer's belief that the module is indeed significant.

In our algorithm, the random walks are performed over the *concept graph* extracted from the program source. And the random walk algorithms underpinning the reinforcement and the propagation judgements are adapted from two well known random walk models: PageRank [17] and HITS [18]. In the next sections, we formally describe the *concept graph* and the computation model of the random walks.

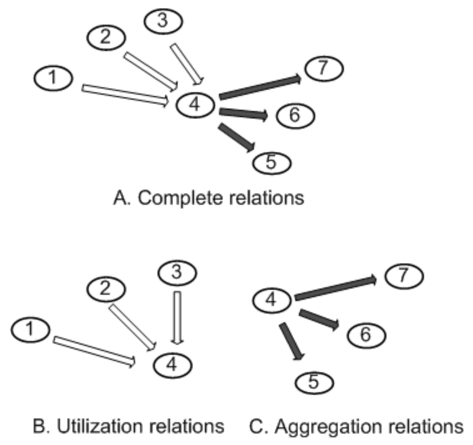


Fig. 2. Relations of modules

B. Concept graph

Our algorithm first uses the concept graph to represent the *compositional* relationships between modules as defined by the developers. Since we are tracking composition and decomposition relationships, our graph is based on the context insensitive call graph and does not require flow or context sensitive information. We define the concept graph as $G =< V, E >$, where V is a set of vertices, $\{c_1, c_2, \cdots, c_n\}$, each of which is mapped to a collection of one or more modules such as packages, classes, methods, or a user-defined group of elements. Each directed edge, $e_{pq} \in E$, denotes that the element set p syntactically "knows about" the element set q in the following ways:

1) *Type extension*: A module in p is a subtype of a modular unit in q as defined by the type system. For Java, a subtype means either a subclass or an interface implementation.
2) *Method call*: A module in p calls a method, the signature of which is *declared* in q. This is to account for practice of interface-based programming. For instance, when using the *Observer* design pattern, a class `Foo` of the `Observer` type implements the `update` interface to receive notifications. Despite the fact that the class `Subject` calls the `update` method of `Foo`, we consider that the call effectively signifies the relationship between the `Subject` and the `Observer`. We can detect this relationship by tracking the class where the invoked method is originally declared.
3) *Method implementation*: An implementation in the subtype of a class type in p calls a module *declared* in q. Again, taking the *Observer* design pattern as an example, the method `addObserver`, which registers observers with subjects, is represented by an edge on the concept graph between the `Observer` module and the interface `Subject` but not its subtypes that implement the method `addObserver`.
4) *Reference*: A module in p has q as a field or method parameter, or it accesses the static members of q.

Note that, in our definition, we use an abstract term *module* to show that our algorithm is generally applicable to a variety of module definitions. In addition, the concept graph for a particular application is precise as we only use the syntactic information of the source, which must be fully resolved and type-checked for a program to be compiled.

C. Modeling random walks

Given the concept graph, $\mathcal{G}$, the core objective of our algorithm is to compute the two counters that are associated with each node. These counters represent the fractions of the total number of steps taken by the random walks that visit that node and, hence, can be substituted by two random variables with probabilities P_u (utilization) and P_a (aggregation). The afore described random walk is a Markov process where probability of visiting the destination node only depends on the probabilities of visiting the source nodes in the previous step and the transition probabilities. We follow the setup of the PageRank algorithm and express this definition formally in Equation 1.

$$P(v_j) = \sum_{i \neq j}^{n} P(i\_to\_j)P(v_i) \tag{1}$$

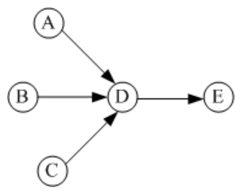


Fig. 3. A simple concept graph of five modules. Arrows represent the direction of coupling between modules.

where $P(i\_to\_j)$ is defined as:

$$P(i\_to\_j) = \begin{cases} \lambda * 1/Outdegree(v_i) + (1-\lambda) * 1/|V| \\ \quad (Outdegree(v_i) \neq 0) \\ \\ 1/|V| \ (Outdegree(v_i) = 0) \end{cases} \tag{2}$$

What Equation 2 defines is the rule for picking the next node to continue the random walk: with λ probability, all outgoing nodes have equal probability to be picked and, with $1-\lambda$ probability, the walk is equally likely to pick any node in the entire graph to continue. For the leaves of the graph, the walk randomly picks any node to continue. To compute all nodes in the graph, we use the matrix form of Equation 2 in Equation 3, where $\vec{P}(v)$ is the probability vector of all vertices and M the transition matrix with each entry $m_{ij} = P(i\_to\_j)$. Note that M is a left stochastic matrix.

$$\vec{P}'(v) = M \times \vec{P}(v) \tag{3}$$

We now present the computation model for both the reinforcement and the propagation judgement random processes. We accompany our discussion with a simple concept graph in Figure 3 to illustrate the characteristics of these two judgement processes and the computation details.

a. Propagation judgement process: The propagation judgement process performs two random walks, one strictly following the utilization direction and the other the aggregation direction. The random walk is performed by repeatedly calculating $\vec{P}_a$ and $\vec{P}_u$ through Equation 3, except that we use M^T instead of M in the calculation of $\vec{P}_a$. It is also a well known fact in linear algebra that the values in $\vec{P}_a$ and $\vec{P}_u$ become stationary as there exists the eigenvalue 1 for both the left and the right eigenvectors of M.

In Figure 4, we illustrate the computation process of the propagation judgement for a toy concept graph in Figure 3. Figure 4 (I) shows that the utilization score of node D, P_D, is computed using the utilization scores of all the incoming nodes A, B, and C. And the node E is computed using the score of node D only. To calculate the aggregation score, we invert the utilization graph and recalculate the transition probilities of edges. Figure 4(II) shows that the aggregation score of node D is obtained using the score of node E only, and the score of node B is calculated from node D only. The dashed arrows signify the probability values, representing the visiting frequencies, "propagate" along the directed edges. Note that the transition probabilities, $P_{AB} \neq P_{BA}$, since the they are calculated separately on two different graphs.

b. Reinforcement judgment process: The reinforcement process takes a step forward in the utilization direction, followed

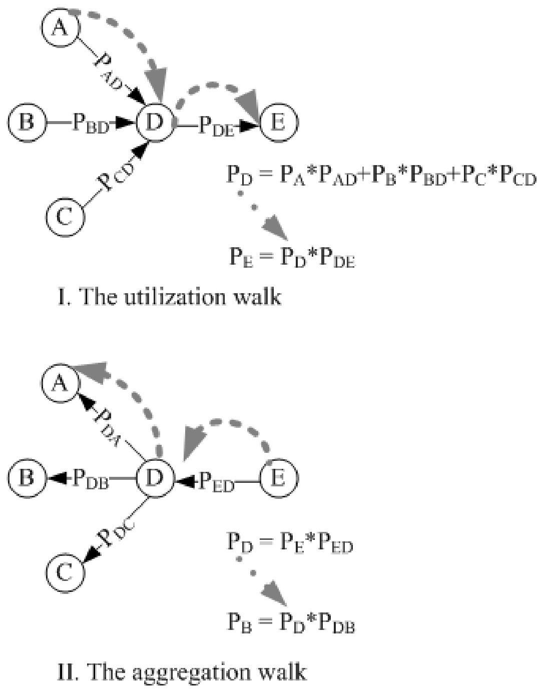


Fig. 4. Propagation judgement. Dashed arrows represent the direction of evaluation.

by a step backward in the aggregation direction. The forward step is equivalent to the matrix multiplication that yields the updated utilization probability vector $\vec{P}_u$. The backward step computes the aggregation vector $\vec{P}_a$ using M^T, the transpose of M. These two steps are expressed by Equation 4. The random walk is carried out by repeating these two multiplications for a large number of times. The fact that the values in both $\vec{P}_u$ and $\vec{P}_a$ become stationary by the repeated multiplications is a standard result of linear algebra, as pointed out by the well known HITS [18] algorithm.

$$\begin{aligned} \vec{P}_u &= M \times \vec{P}_a \\ \vec{P}_a &= M^T \times \vec{P}_u \end{aligned} \tag{4}$$

Figure 5 illustrates that the process of the reinforcement judgement simultaneously computes both the utilization score of node D, U_D, and the aggregation score of node B, AG_B, in one round of random walk. In the forward step, U_D is computed as the sum of the aggregation scores of the incoming nodes multiplied by the transition probabilities. In the backward step, U_D is then used to compute the aggregation score of node B, AG_B. The reinforcement effect comes into play as a larger utilization score for D, U_D, leads to a larger aggregation score for B, AG_B, as these two nodes are mutually dependent on each other.

Our algorithmic design makes a few significant extensions to the original PageRank and HITS algorithms. Compared to the PageRank algorithm, the "ranks", or scores, are mathematically computed in the same way. However, two scores, instead of a single score, are computed for each node on both the concept graph and its inverted version. Compared to the HITS setup, the "hubs" and "authority" scores are calculated using a stochastic matrix. This, however, does not affect the

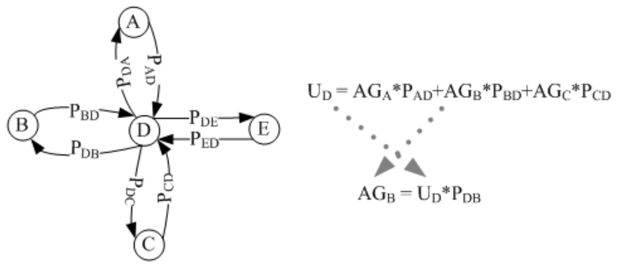
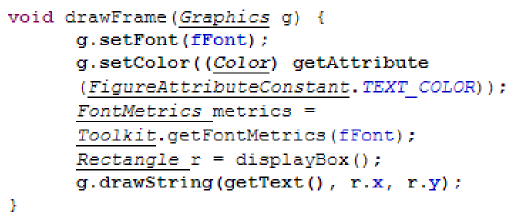


Fig. 5. Reinforcement judgement. Dashed arrows represent the direction of evaluation. U and AG stand for utilization and aggregation probabilities respectively.



```
void drawFrame(Graphics g) {
        g.setFont(fFont);
        g.setColor((Color) getAttribute
        (FigureAttributeConstant.TEXT_COLOR));
        FontMetrics metrics =
        Toolkit.getFontMetrics(fFont);
        Rectangle r = displayBox();
        g.drawString(getText(), r.x, r.y);
}
```

Fig. 6. Biased transition

convergence property of the vectors since $\vec{P}_a' := M \times \vec{P}_u := M \times M^T \times \vec{P}_a$ and $M \times M^T$ is always symmetric non-negative. Please refer to [18] for more details.

D. Adaptations to source code

In designing the computation model, we observe some important differences in traversing program sources compared to traversing hyperlinks in web documents, on which the PageRank and the HITS algorithms are conceived. First, the number of elements in a concept graph of program sources is significantly smaller compared to that of web pages. Second, relations between program elements are usually well defined, compared to the uncontrolled structure of the links in web pages. In reaction to these differences, we have made a few modifications to the canonical setup of the transition matrix M. These modifications are based on our observations and practices.

Small damping factor
The damping factor, λ, in Equation 2, represents the restarting probability, i.e., the probability of the random walker choosing, with equal chances, any other disconnected nodes instead of following a relation link. λ has a significant impact on the ranking results of the underlying graph. For instance, it has been shown [20] that λ is a useful tool for detecting link-spams in ranking web pages. For program sources, we choose $\lambda = 0.05$, a smaller value than the one commonly used, to reflect the reasoning that it is unlikely for a code reviewer to jump from one module to a random one that it has no relation with in the type space. In our algorithm, λ accounts for non-syntactic dependencies between program elements that the miner likely to follow, such as inter-process communications, reflective invocations, or other accidental dependencies that are difficult for the type-based analysis to detect.

Biased transitions
Both the HITS and the PageRank models were initially incepted for analyzing hyperlinks of web pages. In the graph constructions of the original algorithms, there are no weights associated with the edges and each node has equal chance of transitioning to a connected node regardless of the number of actual links between them. This idempotent setup is necessary for resisting malicious manipulations of ranking computations in uncontrolled settings such as web pages. As a CC investigator reads the source code, more references to a particular program element will likely bring that element to her attention, hence, increase its chance of being examined. For instance, when examining the `drawFrame` method of the type `TextFigure` in the Jhotdraw application in Figure 6, an unbiased probability assignment states that the investigator is equally likely to visit 7 other class types[2] due to either method calls or field accesses. Realistically, the investigator would be more biased towards visiting the type `Graphics` compared to other types as the most number of calls (three) are made to it.

To better model the manual process, we use the biased transition where we associate each edge e_{ij} with a weight ω_{ij}[3]. Currently, we define the weight ω_{ij} as the number of method invocations made by module i to module j. The biased transition probability is then defined as:

$$P_{(i\_to\_j)} = \begin{cases} \omega_{ij} / \sum_{k \neq i}^{n} (\omega_{ik}) & if \quad e_{ij} \in E \\ 1/|V| & if \quad e_{ij} \notin E \end{cases} \qquad (5)$$

E. Ranking and classification

Both the propagation and the reinforcement judgement processes produce two vectors, $\vec{P}_u$ and $\vec{P}_a$, each of which assigns a numerical value for every module in the concept graph. To produce the final ranking, we recalculate the utilization vector as $\vec{P}_u{}' = \vec{P}_u / \vec{P}_a$. Consequently, the utilization value of a ranked element is boosted if its aggregation value is small. For instance, for two elements, e_1 and e_2, with equal utilization probabilities, i.e., $P_u^{e1} = P_u^{e2}$, we have $P_u^{e1} > P_u^{e2}$ if element e_1 has a smaller aggregation probability, i.e., $P_a^{e1} < P_a^{e2}$. We output the final ranking according to the corresponding numeric value of each element in the recalculated vector. And we refer to this ranking as the *differentiated ranking*. To quantify the ranking effect of using both the utilization and the aggregation values, we also output the ranks using the numerical values of the original utilization vector, referred to as the *natural ranking*.

F. Example

Now we come back to the motivating example in Section I and illustrate how our algorithm behaves differently compared to the frequency-based fan-in method. Remind that the concept graph of the example is presented in Figure 1. Table I lists the actual values of P_u and P_a, computed by PCM using the propagation judgement, the confidence ratio (l), and the fan-in

[2]Five types are underlined and two others are the type of `fFont` and the return type of `getText`

[3]Similar techniques of weight assignments are also used in analyzing web links [21]

values of the class types. We used the natural ranking method and the biased probability calculation.

From Table I, it is clear that, judging by the usage frequency (fan-in), the top CC candidates are *Customer*, *Logger*, *Connection* and *Storable*. As previously discussed, the candidates *Customer* and *Connection* belong to the essential functionality of the telephone application and cannot be classified as crosscutting concerns. In addition, the candidates *StorableOutput*, which carries out the actual logging functionality, and the classes *Billing* and *Timer*, the two crosscutting concerns in the original distribution, are not reported due to their low usage frequencies. In contrast, our random walk algorithm makes a few different decisions, which we believe are much more reasonable. First, it gives *StorableOutput* the highest utilization value because it is part of the logging functionality. In using frequencies, it is ranked the last. Second, the aggregation probabilities place an important role in deciding that, despite having high fan-in values, *Customer* and *Connection* also aggregate significant program functionality, in coherence with the design of this application. As the result, by using the ratio between utilization (P_u) and aggregation (P_a), PCM reports the top CC candidates as *StorableOutput*, *Loggger*, *Timer*, *Storable* and *Billing*, which include all of the crosscutting concerns in this application.

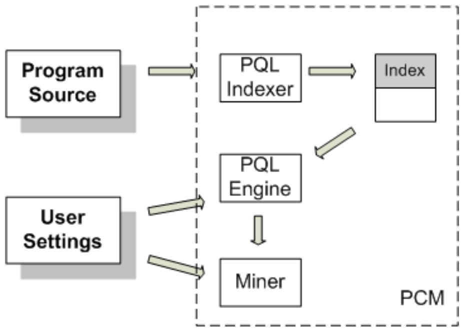


Fig. 7. High-level architecture of PCM

Query	*Specification*
Retrieve component names	`match component:"∧.⋆$";`
Compute a call map for component "X"	`callrootmap(match type:"⋆.⋆" within component:"X");`
Retrieve types of which the methods are invoked by type "Y"	`pick type:"⋆.⋆" outof totype(match call:"⋆.⋆.⋆(..)" within type:"Y");`
Retrieve subtypes of type "Z"	`match type:"Z+".`

TABLE II
PQL EXAMPLES

III. THE PRISM CONCERN MINER

We have implemented the PCM algorithm as an Eclipse[4] plug-in to enable the ease of use and the seamless integration with the Java development environment. It is publicly available[5], and its high level architectural components are depicted in Figure 7.

Given the program source, usually through the Eclipse project information, PCM first indexes the source using the native Java compiler in the Eclipse JDT[6]. The generated index file is kept in memory to support efficient mining operations. The users of our tool can specify customized mining settings of the mining task, such as the scope (whole project or selected components) of the analysis, or the granularity (method or class) of the module considered. The miner then works with the PQL engine to build the concept graph and to conduct random walks.

The primary purpose of IDE integration is to demonstrate and to evaluate an unique feature of our algorithm, which is to allow the users of PCM to influence its computation process in order to inject a certain degree of their domain knowledge of the application. This domain knowledge injection is accomplished by a query language, PQL, that we have designed for this research. In the following section, we first briefly introduce PQL and the details of the domain knowledge injection supported by PCM.

A. Prism query language

The Prism Query Language (PQL) is a simple declarative query language we have developed for conveniently describing facts about Java systems. Our convenience claim comes from the fact that PQL adopts the AspectJ type patterns with a small addition of scope operators. The underlying pattern matching is also supported by the matching mechanism of the AspectJ compiler. With simple set operators such as "union" or "intersect", users of PCM can quickly describe a set of Java elements that match a certain criterion of type definition patterns and usage patterns. In Table II we give a few examples of the typical PQL statements used in the mining algorithm. A set of APIs in PQL allows the embedding of these statements within Java programs for the dynamic definition of queries. PQL uses memory-based indices to process queries against large code bases with good response time. Detailed information and the executable of PQL are publicly available[7].

B. Expert knowledge injection

In PCM, *Expert knowledge injection* is the capability of customizing how concepts are mapped to the elements in the program source, as a way of injecting the expert knowledge.

PCM supports the following ways of knowledge injections:

1. ***Exclusion*** – For large applications, a human miner is often only interested in investigating parts of the code space. For instance, software packages such as the graphic editor, JHotdraw, often include a large number of sample applications which, albeit not useful in understanding the internals of the JHotdraw framework itself, can skew the mining results significantly, as confirmed by our observations (Section IV-I). The option `ignore`, taking a PQL query as its value, excludes non-interested program elements for a particular run of PCM. For example, the query `match`

[4]Eclipse. http://www.eclipse.org
[5]The Prism Eclipse plug-in. http://www.cse.ust.hk/~charlesz/prism
[6]Eclipse Java Development Toolkit. URL:http://www.eclipse.org/jdt

[7]Prism Query Language. http:\\www.cse.ust.hk\~charlesz\pql

$Concept$	$Aggr.(P_a)$	$Util(P_u)$	$l(P_u/P_a)$	$Freq$	$Rank_{pcm}$	$Rank_{feq}$
StorableOutput	0.005000001	0.245422857	49.08455961	1	1	10
Logger	0.006295456	0.125054376	19.86422839	4	2	2
Timer	0.006993562	0.075360962	10.77576199	1	3	10
Storable	0.006727274	0.058618246	8.713521042	3	4	3
Billing	0.041578541	0.058253402	1.40104487	1	5	10
Connection	0.080870607	0.09904377	1.224719001	4	6	2
Local	0.04300714	0.034931602	0.812227974	1	7	10
LongDistance	0.04300714	0.034931602	0.812227974	1	7	10
Customer	0.377310965	0.143019241	0.379048726	5	9	1
Call	0.389209312	0.125363941	0.322099029	1	10	10

TABLE I
EXAMPLE

`type:"org..samples.*"` filters out all sample code shipped with JHotdraw version 6.

2. ***Specialization*** – Opposite to exclusion, the `select` option can be used to narrow the scope of processed elements. This is analogous to search engines combining ranks with a certain type of context such as keywords or locality. For instance, the PCM user can produce rankings only for subtypes of `Figure` by using the PQL query: `match type:"*..Figure+"`.

3. ***Customization*** – The default concern types understood by PCM are module types defined in the Java language such as *method, class*, or *package*. However, concerns do not always have to align with the boundaries of modules. Instead, they can be mapped to patterns in the type space. For example, the concept of *figure element* covers all subtypes of the type `Figure`. In the JHotdraw 6 distribution, these types span four different Java packages. Concerns can be mapped to composition patterns. For example, the concept of *networking layer* can be defined as all types having fields of type `Socket`. Concerns can also be mapped to interaction patterns such as defining the concept of *Event generator* to be types invoking the `fireEvent` method. PCM is capable of provisioning these three kinds of *user-defined concepts* by taking key-value pairs which associate an unique concept name with a pattern defined in a PQL query. This name, representing the user-defined set, is used by PCM in the ranking evaluations on behalf of the actual data types contained in the set. Let us use a simple example to illustrate how customization could improve the mining results in the case of JHotdraw.

Our initial user study of knowledge injection shows that the process of concept customization typically undergoes a series of iterative refinements: the user usually starts with the general ranking to get an overview of the CC candidates; she then defines a search query with respect to the interested program elements to browse the code-level footprints of the crosscutting phenomenon; based on the code-level exposure, she might choose to refine the search query to more accurately represent the investigation target; she then uses the refined query to produce a set of new rankings. We believe that these iterative steps are best integrated with the development environment, and we have created the Prism Eclipse plug-in to allow Java developers to perform mining directly on their project sources.

IV. EVALUATION

To quantify the effectiveness of our algorithm and to properly compare to the related mining approaches, we conduct a large array of experiments which aim to answer the following questions.

1. ***What is the quality of the results compared to the manual effort?*** Due to the semantic nature of CCs, we believe that the judgement of domain experts, given sufficient time, should always be superior to an algorithmic one in determining if a feature is a crosscutting concern. Therefore, comparing how closely algorithmic results is to the expert judgement is an effective way to assess its quality. We quantify the performance of the CC algorithms by comparing to the our earlier [6], [19] extensive manual CC investigation on a middleware application, conducted prior to this research for a different and independent research purpose. Our conclusion is that PCM can generate comparable results compared to our months of manual classification effort.

2. ***What is the performance of PCM compared to the related approach?*** To compare to the state of the art, we choose the FINT [7] tool as the representative technique. To enable a fair comparison, we carefully constructed an unbiased oracle. We collect a set of features in the JHotdraw application, which are reported as CCs in the literature. Using this oracle data set, we observe the PCM is significantly more effective, from 40% to as much as 3X, in both recall and precision compared to FINT.

3. ***What is the difference between the reinforcement and the propagation judgement?*** We evaluate both the propagation and the reinforcement judgements and also experimentally discuss their differences. Our general conclusion is that the reinforcement judgement produces better results as compared to the propagation judgement in general. In the JHotdraw benchmark, the propagation judgement performs better for top ranked results.

4. ***How effective is PCM in analyzing new applications?***. To test the effectiveness of PCM on unknown applications, we worked with domain experts of a third-party application and ask the developers to verify the correctness of the findings of our algorithm. We found that PCM was able to locate latent features which were confirmed as previously unknown crosscutting concerns.

5. ***Can PCM efficiently work with large software systems?*** We quantify the scalability of PCM algorithm in processing

software systems ranging from a few thousand lines of code to over 2 million lines. We observe that the runtime of PCM scales well with respect to the size of the mining target.

We now discuss our experiments in details.

A. Experiment methodology

Similar to conventional information retrieval approaches, our basic quantification method is to benchmark the performance of the mining algorithms against objectively constructed oracle data sets. However, to the best of our knowledge, there does not exist a well verified and recognized benchmark for measuring the quality of CC algorithms. Our oracle data sets are constructed at least unbiased with respect to all the mining algorithms that we evaluate. We use two data sets in our evaluation. Our first data set, orbacus, is the result of our independent research effort prior to the inception of the mining algorithm. Our second set, jhotdraw, is constructed using the non-discriminative syntactic matching on program elements. We evaluate the quality of the algorithms by commonly used information retrieval metrics such as *precision*, *recall*, and F_1 measure for the Top-K returned results judged by our oracle data sets. This is a standard evaluation method for ranking-based statistical algorithms like ours [22]. In our evaluation, we compare both natural and the differentiated ranking methods to quantify the effect of using both the bi-directional and the transitive properties, the two main insights used in our algorithm. The incremental interval of K is picked at 25, a value we feel can properly and succinctly reflect the properties of the compared algorithms.

B. Experiment 1: compare to manual effort

Our first experiment is to compare the PCM results to quantify the precision and the recall with respect to our manual classification of CC concerns in the source of the ORBacus middleware. We emphasize the following facts about this experiment. First, the oracle data set, i.e., our manual classification that we compare to, is not deliberately constructed for this study. Instead, the date set is the result of a two-year independent study [23], [6], [19] of CCs in middleware systems prior to the inception of the PCM algorithm. Therefore, the classification is not biased towards the PCM algorithm. Second, due to the semantic nature of crosscutting concerns, we believe that the manual classification based on domain knowledge is always superior to algorithmic conclusions. Our study is to reflect how close the automated mining is to our manual effort. By this reason, we will not compute the F measure in this experiment. We now present the oracle data set and the evaluation details.

1) Oracle data set: ORBacus is IONA's CORBA[8] product. The entire ORBacus distribution consists of around 1800 Java class types. Our manual classification produces a CC set consisting of 557 class types. Our previous research results [23], [6], [19] indicate that these classes not only crosscut the core structure of the application and, as a noteworthy characteristic, the remaining classes, after them being refactored out, actually comprise a runnable version of the original application with the minimal core features [19]. We note that this data set reflects our research experience as CC experts and might not be representative of other large applications. Nevertheless, it is the largest and thoroughly validated "aspectized" application available to us.

To focus on class types that represent the design effort of programmers, we exclude, from the reference CC set, the *support code*, i.e., the auto-generated code included in the ORBacus source distribution, which is not part of the functional implementations of ORBacus itself. These types include "stubs", "helpers", and "holders". We also include the *interface types* into the CC set. It is common to define the object behavior as interfaces for conforming classes to implement. Such kind of interfaces, post-fixed by the string "Operations" in ORBacus, can be treated as a form of crosscutting concerns [24].

2) Evaluation and observation: In Figure 8, we report the precision of the top k results recommended by PCM as compared to our manual data set. We vary k in steps of 25 from 25 to 525, an upper bound close to the size of the oracle data set. We evaluated both the propagation and the reinforcement judgements, with and without the use of differentiation. We make the following observations of the results.

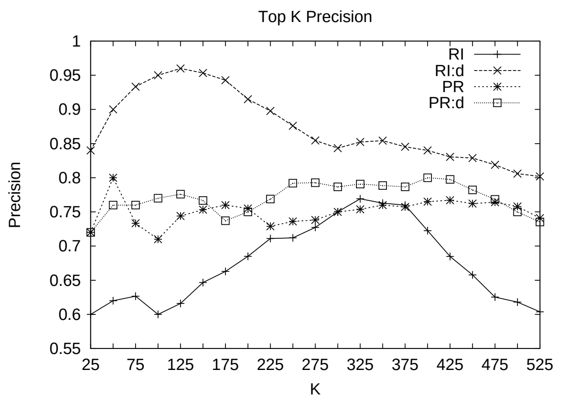
1. On average for all the K values, the precision improvement of PCM ranges from 20% (RI) to 56% (RI:d) and the recall from 22% (RI) to 53% (RI:d). The best precision is obtained by the differentiated reinforcement judgement at around 95% for the top 125 results. The quality of the Top-K results starts to consistently deteriorate as k increases. This method also achieves the best recall at around 75% for top 515 results at the precision of 81%. We believe this performance is comparable to the our manual classification effort. The best recall is also achieved by the differentiated reinforcement judgement at around 72% with 80% accuracy.

2. The differentiation method generally improves the ranking quality of the both judgements. The reinforcement judgement is more sensitive to the differentiation method as it boosts the precision of the top-K ranks by 30% to 61%. The improvement on the propagation method is more significant as the window size gets large.

3. As we increase the size of the ranking window, the number of correctly recalled elements also increases for both judgements. The precision of the propagation judgement shows very limited variations. However, the changes in the case of reinforcement exhibit "hill"-shapes. This shows that the reinforcement method clusters CC candidates and, however, assigns them low absolute utilization values. After being boosted by the differentiation, this clustered is shifted upwards to the range of between 200 and 400 to top 200.

3) Hypothesis testing: From the statistic point of view, one can be curious of how likely the precision computed by PCM is by chance, i.e., equivalent to the random sampling of the data set. To clarify this hypothesis, we pick $k = 125$ where the number of correctly identified CC candidates, n, by PCM are 77 (RI), 119 (RI:d), 92(PR) and 96(PR:d). To calculate the probability of retrieving these elements by

[8]Common Object Request Broker Architecture. URL: http://www.omg.org/corba



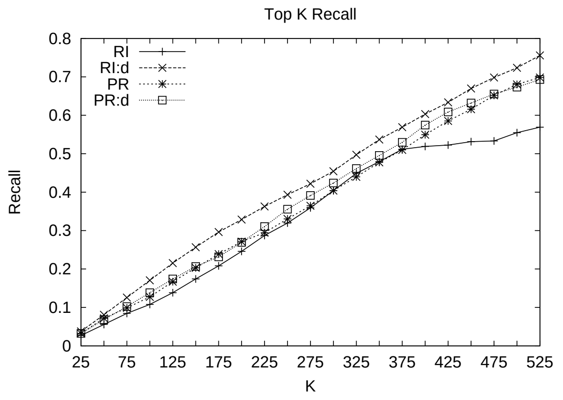


Fig. 8. Precision and Recall. *RI* Reinforcement *RI:d* Reinforcement with differentiation *PR* Propagation *PR:d* Propagation with differentiation

chance, we calculate the total number of equivalent samples, each consisting of n elements sampled from our oracle data set and $k-n$ elements from the rest of the data. We then divide it by the total number of samples of size k from the entire populations. Following the formula, $\binom{557}{n} \times \binom{1815-557}{125-n} / \binom{1815}{125}$, it is easy to verify that obtaining the results by random sampling is very unlikely by chance.

C. Experiment 2: compare to the fan-in approach

In this section, we conduct a comparison study between PCM and FINT [7], a representative frequency based CC mining tool. We use two data sets to quantify the performance differences between FINT and PCM: the ORBacus data set and the JHotdraw[9] data set, consisting of 400 method definitions out of around 3700 methods in total. The ORBacus data set is the same as in the previous experiment. To generate the JHotdraw data set, we first selected the following features reported as CC by FINT: *observer, undo, persistence, visitor*, and *command*. We then issue PQL queries to extract all the method definitions pertaining to these five features[10]. Since PQL queries are only declarative descriptions of program features, our oracle data set is independent of any specific mining algorithms.

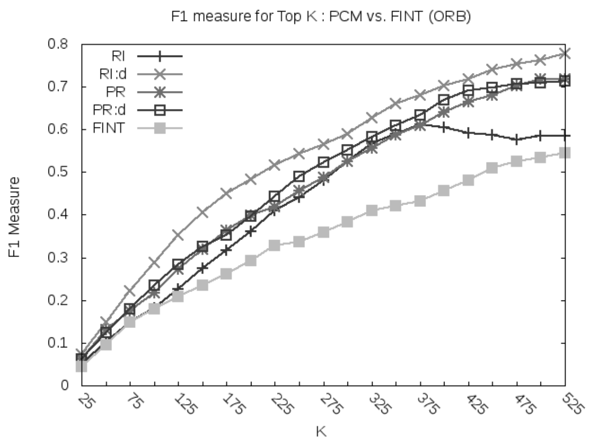


Fig. 9. Comparison of FINT and PCM (ORB)

In Figure 9, we compare the F_1 measure of the Top-K results between FINT and all options of PCM, using the ORB dataset described earlier. Evaluation based on a single value such as the F_1 measure allows us succinctly compare the results of FINT with all of the options of PCM. The version of the FINT tool available to us does not support ranking at the class level. We therefore faithfully implemented the algorithm based on the FINT publication by ranking all classes according to their degrees of fan-ins. Our observation is that, when $K < 100$, the reinforcement judgement without differentiation performs at par with that of the FINT approach. Meanwhile, all other three options perform better ranging from 28% (PR) to 57% (RI:d). On average, PCM is about 21% (RI), 34% (PR), 38% (PR:d), and 56% more effective than FINT.

In Figure 10, we plot the F_1 measure of Top-400 returned elements for both FINT and PCM on the JHotdraw dataset. The FINT result is obtained directly from the FINT eclipse plug-in released by its authors [7]. We can run PCM at the method granularity and produce global rankings for the two judgement options. Our results show that, for most of the K values, PCM is least two times as effective as compared to FINT. The maximum number of correct elements retrieved by PCM is 120, 3 times as many as that of FINT. On average, PCM is 236% (RI:d) and 233% (PR:d) more effective than FINT as shown by our results. In Table III, we present a micro perspective and show the results of the first 25 methods return by both FINT and PCM. For the first 20 methods, FINT did not return any matching results with respect to the oracle data set. Meanwhile, PCM performs much better, e.g., returning 9 correct methods in the top 10 returned results by the use of the propagation judgement.

D. Experiment 3: the domain knowledge injection study

To study the effect of the domain knowledge injection, we performed a case study on how it influences the ranking quality on the source code of JHotdraw. In Table IV, we compare the top-15 ranked elements computed using plain Java types (2nd

[9]We used version 6 of JHotdraw from http://www.jhotdraw.org

[10]The details of the PQL queries can be found on the PCM web site: http://www.cse.ust.hk/~charlesz/pcm

K	*5*	*10*	*15*	*20*	*25*
FINT	0	0	0	0	1
PCM (RI)	1	7	10	12	13
PCM(PR)	2	9	11	12	15

TABLE III
RECALLED ELEMENTS ON JHOTDRAW

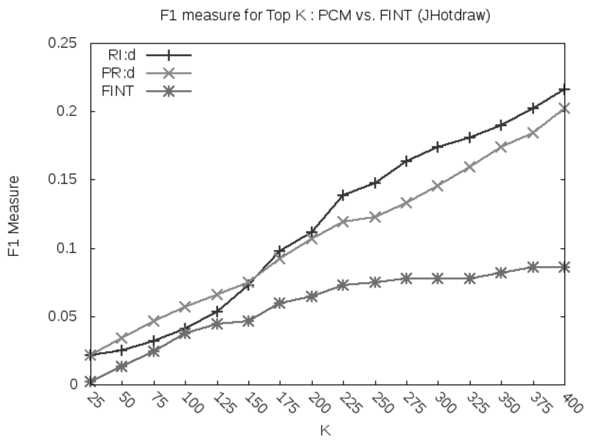


Fig. 10. Comparison of FINT and PCM (JHotdraw)

Rank	*Plain*	*Customized*
1	Figure	CollectionsFactory
2	DrawingView	JHotDraw-RuntimeException
3	Storable	***persistence***
4	TextHolder	Figure
5	FigureEnumeration	DrawingView
6	Undoable	***undo***
7	UndoManager	FigureEnumeration
8	DrawingEditor	DrawingEditor
9	Handle	Handle
10	JHotDraw-RuntimeException	Locator
11	StorableInput	Tool
12	StorableOutput	Command
13	Tool	ConnectionFigure
14	CollectionsFactory	HandleEnumeration
15	Command	Drawing

TABLE IV
EFFECTS OF USING CUSTOMIZED CONCEPT DEFINITIONS

column) to using concept customizations (3rd column). We observe that the types representing the features "persistence" and "undo" are repeatedly reported in the plain ranks. For PCM users wishing to treat "persistence" or "undo" uniformly, this could introduce inconvenience or "noise" when inspecting the top-ranked elements.

To produce more compact rankings, we define two customized concepts *persistence* and *undo* as PQL query statements as follows:

```
persistence --> match type:"*..Stora*"
undo --> match type:"org..Undoable" or
type:"org..UndoManager";
```

The features `persistence` and `undo` are evaluated directly by PCM and ranked as highly crosscutting concepts (placed 3rd and 6th in the 3rd column). In addition, the customized computation gives much more diversified results for the top-ranked elements as it brings more distinct elements under the attention of the PCM user.

E. Experiement 4: the AspectJ compiler case study

In this experiment, we conduct a case study of the CC mining in the source code of the AspectJ [25] compiler where we work with the developers to assess the usefulness of PCM. The version of the AspectJ compiler that we studied is 5.0, consisting of approximately 1000 classes, excluding code written in Java 5 and the entire AspectJ Eclipse plug-in (AJDT). The propagation judgement is used and the mining result is publicly available[11]. To assess the quality of the mining results, we consulted with the AspectJ developers. Our first attempt processes every inter-class relation. The AspectJ developer reported that a large number of top-ranked elements were identified as noise. These elements are well-localized in two packages: `bcel` and `weaver`, which occupy 30% of the total number of class types of the mined AspectJ sources. In a second attempt, to look for "compiler-wide" crosscutting candidates, we only record the classes if the relations between two classes are defined in different packages[12]. We confirmed with the domain expert that the quality of the ranks has improved and a few unexpected but correct CC candidates were also identified. We list some examples of the resulting CC candidates specific to the AspectJ compiler discovered by PCM:

Structural model: The structural model maintains the information about the structure of both aspects and classes as well as the relations between aspects and advised classes. The support for the structural model are implemented in the AspectJ compiler in various places such as building component, UI support, and JavaDoc functionality.

Backwards compatibility of the weaver: AspectJ aims to support backwards compatibility so that a newer version of AspectJ can load aspects compiled with an older version. To enable this, the version information is stored as standard Java class file attributes. The support of these attributes is scattered across the weaving component

Compile and weaving context: The compile-and-weave context "*is responsible for tracking progress through the various phases of compilation and weaving*". It is used to create a "stack trace" that "*gives information about what the compiler was doing at the time*"[13] when unanticipated events occur during the compilation and the weaving process.

[11]AspectJ mining results. URL:http://www.cse.ust.hk/~charlesz/pcm/aj.txt

[12]This can be set as an option in our plug-in.

[13]See comments of the class `CompilationAndWeavingContext` of the AspectJ 5.0 source. URL: http://www.eclipse.org/aspectj

F. Large scale CC mining

In this experiment, our mining target is the IBM Websphere Application Server 5.0 source code. The whole of the Websphere Application Server 5.0 code base consists of around 15K classes, 3 millions lines of code, and more than 125 independently built components. Not all of the sources are compiled for a particular build instance. In our instance, PQL has indexed over 8000 classes comprising approximately 2.1 million lines of code. The version built with PQL indices passes the relevant build verification tests.

Correct classifications The crosscutting functionalities in the Websphere Application Server reported by Colyer and Clement from their manual investigations include the *Websphere diagnostics and serviceability* components, the *Websphere performance monitoring infrastructure*, and the *WebSphere EJB container* component. These components are also captured by our algorithm (ranked 2nd, 3rd, 12th, and 15th). The *Websphere security public interface* component (ranked 9th) and *transaction service public interface* component (ranked 21th) contain interfaces for accessing the security and the transaction support, hence, represent typical crosscutting concerns for enterprise systems.

Misclassification's: As expected, our top-20 ranking produces false positives for pivotal building blocks of the system. For instance, the *Websphere user interface* component implements the base functionalities of the browser-based administration console for Websphere. This is misclassified as it is an essential component for more specific console applications. Same misclassification happens to the *Websphere component framework* which provides runtime support for about 26 other components. The *WebSphere shared utility* component, comprising many utility functions, is also a misclassification. Utilities are often general and yet fundamental computations of the application logic. They are often essential to the functionality of the application and cannot always be classified as crosscutting concerns.

Surprises: Caching and logging are typically referred to as aspects. One would expect to find them in an application server. In the Websphere Application Server, the *WebSphere caching* component is responsible for improving the response time of `Servlets` by caching their results. However, this component is strongly classified as a non-crosscutting component (receiving the lowest ranking in the CC ranks and 10th in the non-CC ranks). Upon examination of the source code, this caching functionality is indeed well modularized as an interceptor to intercept "calls to cacheable objects, for example, through a servlet's `service()` method or a command's `execute()` method" [26]. The *Websphere commons logging* component, which can easily be mistaken as a classic crosscutting concern, is also reported as non-crosscutting by our algorithm. This component is not responsible for the actual logging functionality in the Websphere Application Server. It is an adaptation of the Apache logging interface with the native Websphere Application Server logging functionality in the Websphere servicability component.

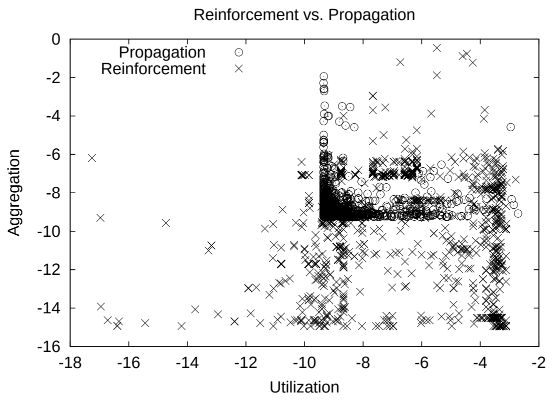


Fig. 11. Propagation vs. Reinforcement

G. Choices of PCM parameters

There are two major parameters of PCM: the method of ranking (*natural or differentiated*) and the type of the judgement (*propagation or reinforcement*). We now briefly discuss the observed impact of these parameters based on the experiments on the ORB and the JHotdraw datasets.

Judgement type For both the ORB and JHotdraw data sets, the Top-K performance of the reinforcement judgement is generally superior to that of the propagation judgement. To obtain additional insights on the differences of these two random walk models, in Figure 11 we plot the logarithmic values of computed utilization and aggregation probabilities for both judgements on the ORB data set . It can be observed that the range of changes in the case of the propagation judgement is much smaller compared to that of the reinforcement, as most of the propagation values are clustered towards the upper right quadrant of the graph. It shows that the probability pairs computed by the reinforcement method exhibit larger differences. And this polarization supports the phenomenon that applying the differentiation method causes a significant boost to the rankings in the ORB data set.

Ranking method The differentiated ranking method has various degrees of improvements on the ranking quality of PCM on the ORB data set. For top-100 elements, it improves the quality of the reinforcement judgement by 30% and of propagation by 3%. Since its application to the reinforcement judgement produces the best results on our data sets, it is the default ranking method of PCM.

H. Runtime characteristics

To quantify the efficiency of PCM we have chosen 7 Java applications of various types and sizes: graphical editing (i.e., JHotdraw), databases (Prevayler[14], hSQL[15], Derby[16]), middleware implementations (ORBacus, Websphere Appli-

[14]Prevayler. URL: http://www.prevayler.org
[15]HSQL Database Engine. URL: http://www.hsqldb.org/
[16]Apache Derby. URL:http://db.apache.org/derby/

	No. of Types	Duration(sec)	LOC
Prevayler	38	0.15	2,396
Padres	203	1.04	17,124
HSQL	310	1.77	48,300
JHotdraw6	398	1.59	15,541
Derby	1,261	15.9	153,000
AJ1.5	1,353	11.67	89,634
ORBacus	1,815	24.1	64,704
WAS	8,800	1,085	2,000,000

TABLE V
MINING PERFORMANCE

cation Server, and PADRES[17]) and the AspectJ compiler version 5[18].

We measure the size of the application, both in terms of number of class types and lines of code (LOC), as well as the time for PCM to generate the mining results. The PQL indexer incurs negligible runtime overhead for the normal compilation process. All experiments are performed on an IBM ThinkCentre workstation running the Linux 2.6 kernel on a Pentium 4 CPU at 3.2G Hz with 1.5G of physical memory. The maximum heap memory used by PCM is set at 512M for all experiments except for the WebSphere application server, where it is set to be 1.5G. Table V shows that for most mid-sized applications, PCM is able to produce the results in less than 30 seconds. Scaling up to large scale applications such as the Websphere Application Server, PCM requires 18 minutes to complete on our workstation (a conventional desktop PC[19].)

I. Lessons learned

From our experiments and observations, we summarize some typical mis-classifications to serve as guidelines for interpreting PCM-computed results. We attribute most of these mis-classifications to the accidental crosscutting phenomenon. That is, core concerns *syntactically* crosscut the code base for the following reasons:

Fundamental building blocks: Certain types are widely referenced, serving as fundamental building blocks of the system. These types themselves are simplistic in terms of collaborating with other types. Examples of such types include `Figure` in JHotdraw and `Buffer` in ORBacus. These types often appear in the top ranks due to the low aggregation values. It is not difficult to filter out these false positives with the domain knowledge of the application at hand.

Support code: The presence of significant pieces of support code, including both added functionalities and samples of demonstrations, can cause key components of the system to syntactically scatter. Examples of such types include sample applications included in the JHotdraw distribution and the skeleton code in ORBacus. Additional treatment is needed to exclude these program elements from skewing the ranking results. Our experience is that this is easy to do since these types of program elements usually follow a certain naming convention. The expert knowledge injection capability of PCM can be leveraged to achieve this.

Utilities: It is common practice to group general computation logic into so-called ”utility” types such as in the `org.jhotdraw.util` package of JHotdraw. Utility types are difficult to even classify manually because their functionalities, such as bit flipping or searching, are often fairly independent of the application itself. These types typically receive high CC rankings and can be easily identified by the miner. This phenomenon is also identified by other researchers [7].

In addition to mis-classifications, the ranking results can be skewed due to local crosscutting in large packages as in the case of our AspectJ experiment. Qualifying the package level crosscutting in our algorithm can effectively reduce local noise. However, the useful information might be lost and lower qualification levels are still necessary if the natural modular boundaries are not fine-grained, i.e., in applications containing super packages or so-called “God” classes.

V. RELATED WORK

Research in the area of aspect mining and CC discovery can be roughly classified into three categories: static analysis, runtime analysis, combined analysis, and multi-modal analysis. The first two categories are based on the program itself, and the last category relies on other artifacts related to the program inspected. Due to the absence of benchmarks, it is difficult to offer a quantitative comparison of the quality of all approaches. Our comparison is thus from the methodological perspective.

Early approaches for CC discovery aim at facilitating the description and the specification of CC to aid the human aspect miner in her concern-discovery-by-query task over large code bases. AMT [27] and AMTex [28] enable the specification of crosscutting concerns using both type and lexical patterns. JQuery [29], CME[20], and PQL provide language-based approaches to improve the expressiveness of this specification. FEAT [30] is based on recording the code browsing and code manipulation process to track and map concerns. All these approaches are more manual and query-based in nature, they do not fully automate the actual discovery of crosscutting concerns. The strive for more automation in CC discovery is the main objective driving the algorithm developed in this paper.

Early automations of CC discovery are based on analyzing program element frequencies and exploiting the syntactic homogeneity of crosscutting concerns. Marin, Deursen, and Moonen [7] carried out a fan-in analysis on various systems to account for the “popularity” of crosscutting types. Bruntink *et al.* [8] presented the detecting of scattered code clones for locating crosscutting concerns. Our earlier work has introduced the notion of “degree of scattering” [6] to produce ranks of frequently used types and methods in Java systems. Compared to this class of approaches, our random walk based algorithm

[17]http://padres.msrg.toronto.edu/

[18]http://www.eclipse.org/aspectj

[19]On the same workstation, the build process for a Java application comprised of about 10,000 classes and 2.4 million lines of code takes about 1 hour.

[20]Concern Manipulation Environment. URL: http://www.research.ibm.com/cme/cme

presented in this paper is able to reflect the "network effect" of program elements. Our simultaneous use of "utilization" and "aggregation" values provides additional rationals for classifying mining targets. In addition, compared to earlier approaches, we also address the discovery of heterogeneous crosscutting concerns.

Numerous approaches have been dedicated to the runtime analysis of programs. Tonella *et al.* [10] demonstrated the effectiveness of using formal concept analysis over execution traces of a program. Breu *et al.* [31] also exploits execution traces in the DynAMiT framework to discover crosscutting concerns. Execution traces are effective in overcoming the semantic barrier often encountered in syntactic analysis. Compared to these approaches, our approach is syntax-based operating directly on the program sources.

Researchers have also shown that, by combining several sepecialized techniques [32], [15], the mining quality is better than using each of them individually. We classify PCM as a specialized technique that can also be added to the repertoire of these combinational approaches.

Muli-model analysis means incorporating artifacts other than the syntactic information of the program source for the purpose of locating crosscutting concerns. Baldi et al [12] uses the DLA probabilistic model to associate CC with latent topics by treating variable and class names as words. Shepherd *et al.* [33] leverage natural language processing capabilities together with the keywords and comments of the source for clues about crosscutting concerns. Breu et al [11] and Adam et al [14] make use of CVS histories in tracking crosscutting updates. Aspect mining in requirements [34], [35], [36] has also been shown to be an effective approach. These approaches in general complement the source-code based mining techniques like ours.

Inoue *et al.* [37] presented an application of the PageRank algorithm for ranking components in Java program sources. The ranks, interpreted as weights, are equivalent to our popularity ranks used to identify "fundamental and standard" [37] types. Aside from solving a different problem, – the problem of CC discovery,– our algorithm has many significant technical differences. For example, we adjust the PageRank algorithm to reduce the randomness of analyzing program sources having controlled structures. In addition to popularity ranks, we simultaneously use significance ranks to reflect the properties of components in another dimension.

VI. Conclusion

We have proposed the use of random walks to approximate the process of how a human miner of crosscutting concerns distinguishes between core elements and crosscutting concerns without knowing about the application semantics. The goal of these random walks is to calculate probabilistic values that characterize the degree of utilization and aggregation for each CC candidate under investigation. We implemented our algorithm as the Prism CC Miner (PCM) Eclipse plug-in and made it publicly available for evaluation. Leveraging the flexibility and the efficiency of the Prism Query Language, users of PCM can also influence the behavior of the algorithm by customizing how nodes of the relation graph are mapped to the elements in the program sources. We have evaluated PCM extensively on various oracle data sets as well as through individual case studies with domain experts. Our quantifications show that PCM can achieve comparable performance compared to the manual classification effort. It is capable of identifying hundreds of crosscutting concerns with 95% accuracy. As compared to a representative frequency based mining approach, the performance of PCM is also significantly superior.

Acknowledgments

The authors wish to express deep appreciations to Andrew Clement, Matt Chapman, and Matthew Webster of IBM UK for supporting this research. Special thanks to Andrew Clement for the technical help and insights and to Julie Waterhouse of IBM Canada for provisioning the project and reviewing an early draft of this paper. We also thank Subbarao Meduri of IBM US for the technical verification of the paper. Thanks to Adrian Colyer who has played the crucial role of initiating and provisioning the project. The authors are also extremely grateful for many insightful comments from Prem Devanbu that helped improve this manuscript significantly.

This research has been supported in part by an NSERC grant, by an IBM PhD fellowship for the first author, and by Hong Kong RGC GRF grant . The authors are very grateful for this support.

References

[1] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-oriented programming," in *ECOOP*, Mehmet Akşit and Satoshi Matsuoka, Eds., Berlin, Heidelberg, and New York, 1997, vol. 1241, pp. 220–242, Springer-Verlag.

[2] Marc Eaddy, Thomas Zimmermann, Kaitlin Sherwood, Vibhav Garg, Gail Murphy, Nachiappan Nagappan, and Alfred Aho, "Do Crosscutting Concerns Cause Defects?," in *IEEE Transactions on Software Engineering*. 16 May 2008, IEEE Computer Society.

[3] Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen, "Towards just-in-time middleware architectures," in *AOSD05*, New York, NY, USA, 2005, pp. 63–74, ACM Press.

[4] Frank Hunleth and Ron Cytron, "Footprint and Feature Management using Aspect-Oriented Programming Techniques," in *Languages, Compilers, and Tools for Embedded Systems (LCTES'02)*, 2002.

[5] Adrian Colyer and Andrew Clement, "Large-scale AOSD for middleware," in *3rd International Conference on Aspect-oriented Software Development (AOSD'04)*, Lancaster, UK, 2004, pp. 56 – 65.

[6] Charles Zhang and Hans-Arno Jacobsen, "Refactoring Middleware with Aspects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 11, pp. 1058–1073, November 2003.

[7] Marius Marin, Arie van Deursen, and Leon Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM TOSEM*, vol. 17, no. 1, 2007.

[8] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourw, "On the use of clone detection for identifying crosscutting concern code," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, 2005.

[9] Silvia Breu and Jens Krinke, "Aspect mining using event traces," in *ASE*, Washington, DC, USA, 2004, pp. 310–315, IEEE Computer Society.

[10] Paolo Tonella and Mariano Ceccato, "Aspect mining through the formal concept analysis of execution traces," in *WCRE*, Washington, DC, USA, 2004, pp. 112–121, IEEE Computer Society.

[11] Silvia Breu and Thomas Zimmermann, "Mining aspects from history," in *IEEE ASE*, Sebastian Uchitel and Steve Easterbrook, Eds. September 2006, ACM Press.

[12] Pierre F. Baldi, Cristina V. Lopes, Erik J. Linstead, and Sushil K. Bajracharya, "A theory of aspects as latent topics," in *OOPSLA*, New York, NY, USA, 2008, pp. 543–562, ACM.

[13] Danfeng Zhang, Yao Guo, and Xiangqun Chen, "Automated aspect recommendation through clustering-based fan-in analysis," in *ASE*, Washington, DC, USA, 2008, pp. 278–287, IEEE Computer Society.

[14] Bram Adams, Zhen Ming Jiang, and Ahmed E. Hassan, "Identifying crosscutting concerns using historical code changes," in *ICSE*, New York, NY, USA, 2010, pp. 305–314, ACM.

[15] David Shepherd, Jeffrey Palm, Lori Pollock, and Mark Chu-Carroll, "Timna: a framework for automatically combining aspect mining analyses," in *ASE*, New York, NY, USA, 2005, pp. 184–193, ACM.

[16] Charles Zhang and Hans-Arno Jacobsen, "Efficiently Mining Crosscutting Concerns through Random Walks," in *AOSD07*, New York, NY, USA, March 2007, ACM Press.

[17] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, "The pagerank citation ranking: Bringing order to the web," Technical report, Stanford Digital Library Technologies Project, Stanford University, Stanford, CA.

[18] Jon M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, pp. 604–632, 1999.

[19] Charles Zhang and Hans-Arno Jacobsen, "Resolving feature convolution in middleware systems," in *ACM OOPSLA*, New York, NY, USA, 2004, pp. 188–205, ACM Press.

[20] Hui Zhang, Ashish Goel, Ramesh Govindan, Kahn Mason, and Benjamin Van Roy, "Making Eigenvector-Based Reputation Systems Robust to Collusion," in *Third International Workshop on Algorithms and Models for the Web-Graph*, 2004, vol. 3233 of *LNCS*.

[21] Taher H. Haveliwala, "Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, pp. 784–796, 2003.

[22] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze, *An Introduction to Information Retrieval*, Cambridge University Press, 2009.

[23] Charles Zhang and Hans-Arno. Jacobsen, "Quantifying aspects in middleware platforms," in *AOSD03*, New York, NY, USA, 2003, pp. 130–139, ACM Press.

[24] Paolo Tonella and Mariano Ceccato, "Refactoring the aspectizable interfaces: An empirical assessment," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 819–832, 2005.

[25] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, "An overview of AspectJ," in *ECOOP*, 2001, vol. 2072, pp. 327–355.

[26] Carla Sadtler, "Websphere application server v5 architecture," URL: http://www.redbooks.ibm.com/abstracts/redp3721.html?Open.

[27] Jan Hannemann and Gregor Kiczales, "Overcoming the Prevalent Decomposition of Legacy Code," in *Workshop on Advanced Separation of Concerns at IEEE ICSE*, Toronto, Ontario, Canada, 2001, URL: http://www.cs.ubc.ca/~jan/amt/.

[28] Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen, "Extended Aspect Mining Tool," CASCON 2003 Poster. URL:http://www.eecg.utoronto.ca/~czhang/amtex, October 2002.

[29] Doug Janzen and Kris De Volder, "Navigating and querying code without getting lost," in *AOSD*, New York, NY, USA, 2003, pp. 178–187, ACM Press.

[30] Martin P. Robillard and Gail C. Murphy, "Concern graphs: Finding and describing concerns using structural program dependencies," in *IEEE ICSE*, Orlando, Florida, USA, May 19-25, 2002 2002.

[31] Silvia Breu, "Extending dynamic aspect mining with static information," *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, vol. 0, pp. 57–65, 2005.

[32] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourw, "Applying and combining three different aspect mining techniques," *Software Quality Journal*, vol. 14, pp. 209–231, 2006, 10.1007/s11219-006-9217-3.

[33] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *AOSD*, New York, NY, USA, 2007, pp. 212–224, ACM.

[34] Yijun Yu, Julio Cesar Sampaio do Prado Leite, and John Mylopoulos, "From goals to aspects: Discovering aspects from requirements goal models," in *IEEE RE*. 2004, pp. 38–47, IEEE Computer Society.

[35] Américo Sampaio, Ruzanna Chitchyan, Awais Rashid, and Paul Rayson, "Ea-miner: a tool for automating aspect-oriented requirements identification," in *ASE*, New York, NY, USA, 2005, pp. 352–355, ACM.

[36] Lo Kwun Kit, Chan Kwun Man, and Elisa Baniassad, "Isolating and relating concerns in requirements using latent semantic analysis," in *OOPSLA '06*, New York, NY, USA, 2006, pp. 383–396, ACM.

[37] Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto, "Component rank: relative significance rank for software component search," in *ICSE*, Washington, DC, USA, 2003, pp. 14–24, IEEE Computer Society.