Efficiently Mining Crosscutting Concerns through Random Walks *

Charles Zhang and Hans-Arno Jacobsen

Department of Electrical and Computer Engineering,
Department of Computer Science.
University of Toronto
{czhang, jacobsen}@eecg.toronto.edu

Abstract

Inspired by our past manual aspect mining experiences, this paper describes a random walk model to approximate how crosscutting concerns can be discovered in the absence of domain knowledge of the investigated application. Random walks are performed on the coupling graphs extracted from the program sources. The ideas underlying the popular page-rank algorithm are adapted and extended to generate ranks reflecting the degrees of "popularity" and "significance" for each of the program elements on the coupling graphs. Filtering techniques, exploiting both types of ranks, are applied to produce a final list of candidates representing crosscutting concerns. The resulting aspect mining algorithm is evaluated on numerous Java applications ranging from a small-scale drawing application, to a medium-sized middleware application, and to a large-scale enterprise application server. In seconds, the aspect mining algorithm is able to produce results comparable to our prior manual mining efforts. The mining algorithm also proves effective in helping domain experts identify latent crosscutting concerns.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Complexity measures

General Terms Design, Measurement

Keywords Aspect Mining, Aspect Discovery, Crosscutting Concern Discovery

1. Introduction

The goal of aspect mining, or more precisely, the mining of crosscutting concerns (CC) is to detect program elements in legacy applications pertaining to non-modularized coding concerns (for brevity, we hereon refer to these program elements as "crosscutting elements"). Aspect mining is a well defined and actively researched problem far from having a satisfactory solution. An a priori requirement for tools to find crosscutting concerns with high precision is the ability to accurately characterize the crosscutting concern's syntactic traits. Such kind of characterization, we argue, is difficult, if not completely impossible, to obtain. This is because the term "crosscutting" is really a definition about the system's program semantics, something difficult for tools to interpret at the syntactic level.

CCs can be broadly classified, according to their syntactic characteristics, into *homogeneous* concerns, used uniformly in the code space such as in the case of logging, and *heterogeneous* concerns, used non-uniformly, consisting of several different pieces of code scattered throughout the code space [4]. Unfortunately, neither homogeneity nor heterogeneity is sufficient to determine the crosscutting nature of a particular program element. For instance, logging is a typical homogeneous concern. However, the use of `java.lang.String`, a fundamental data type, exhibits the same level of syntactic homogeneity. Domain knowledge is therefore necessary to make the final judgment. As for heterogeneous concerns, one would have to sheerly rely on semantic justifications since these concerns have no syntactic difference compared to non-CC elements. We therefore believe that a more effective solution to aspect mining is going to be probabilistic in nature instead of giving definitive answers.

Our solution is based on modeling the manual process of CC investigation aiming at separating crosscutting elements from those representing the *core functionality* of the system (referred to as *core elements* in the rest of the paper.) In our model, we assume that the human aspect miner has zero domain knowledge about the semantics of the program at hand. His analysis has to be purely based on the syntactic relationships between program elements. For Java sources, such relationships exist in both the type definitions of a class, such as extensions, containments, and associations through method signatures. They also exist in the implementations of types, such as parameter passings, field accesses, and message dispatching.

Our random-walk process is carried out as follows. To locate CCs, the human aspect miner examines the source files and randomly selects a program element c, may it be either class or method, to start. He then proceeds with two rounds of random walks. During the first round, the miner gathers all the program elements, such as class types or methods, that are "known by" the randomly selected module c. Each of the elements, therefore, form an outgoing relation with c. He then observes the fact that the likelihood of visiting the next element e in an outgoing relation depends on how likely the current element is visited and the total number of outgoing relations the current element has. Consequently, the likelihood of visiting the element e, or the "popularity" of e, can be measured recursively considering the popularity of all the incoming elements and the likelihood of visiting e from these elements. The miner then continues his walk by randomly picking an element on the outgoing link and repeats the same observations.

* In the Sixth International Conference on Aspect-Oriented Software Development, Vancouver, British Columbia March 12-16, 2007

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © 2007 ACM 1-59593-615-7/07/03…$5.00

The second round of random walks is in the opposite direction compared to the first. Our miner collects the elements that "know about" the randomly selected module c. He observes the fact that each "know-about" relation shares the contribution of c to the "significance" of the other element, e, in that relation. Similar to the definition of "popularity", the "significance" of an element e is a function of the significance of all the elements that e knows about and how likely the walk takes place from these elements to e randomly following the "know-about" relations. The miner then continues his walk picking an element randomly following an incoming relation and repeats the same measurement. There is also a small chance for the miner not to follow any relations and randomly restart his investigation with a new module. If a module c has no relations to choose from in the direction of the walk, the miner randomly restarts his walk with any other module. In such cases, c has equal contributions to every other module in terms of either its significance or its popularity depending on the direction of the walk.

The aspect miner repeats his walks for an infinite amount of time. For every element in the graph, the miner updates his measurements of popularity and significance. He then makes the following judgments:

1. *Popularity* – If an element is popular, i.e., frequently visited from different elements, the aspect miner likely considers the element to be either a crosscutting element, such as an exceptions, or a fundamental building block of the system, such as figure elements in the Jhotdraw application[1].
2. *Transitive popularity* – Even if an element is *not* frequently visited, it can still be popular if it is mostly visited from popular ones. For example, if a *logger* class is exclusively used as part of the crosscutting exception handling feature, it should still be considered as a crosscutting type despite that it does not syntactically crosscut other parts of the system.
3. *Significance* – If an element references a large number of distinct elements, the aspect miner is likely to consider it more towards a significant element and less likely a CC element. For instance, the type `org.jhotdraw.figures.PolyLineFigure` is significant since it references 43 other class types.
4. *Transitive significance* – Even if an element does *not* reference a large number of elements, it can still be very significant when it mostly uses other significant elements. For example, the class type `org.jhotdraw.samples.nothing.NothingApp` is a core element in the Jhotdraw source distribution because it is a sample application built on top of the Jhotdraw framework. This type uses 15 distinct class types of Jhotdraw. It is still more significant, or less likely to crosscut, compared to the class types in the Jhotdraw framework itself, such as the type `org.jhotdraw.figures.PolyLineFigure`, which references 43 types, almost 3 times as many.

This process of popularity and significance determination closely resembles a probabilistic random walk process. The walk is performed on the coupling graph obtained from the program sources. The nodes on the coupling graph can represent different program elements such as components, packages, classes, methods, or collections of program elements subject to custom definitions. The graph records the afferent (incoming) relations and the efferent (outgoing) relations for each of the program elements. The walks are performed in two directions: The *efferent* direction from roots to leaves, and the *afferent* direction from leaves to roots. The four reasoning rules listed above can be modeled as a Markov-process for the propagation of the probability of an element being visited in the graph. From this Markov model, we can calculate precisely the "popularity" measure for each element as the cumulative probability of it being visited in the outgoing direction, and the "significance" measure of the probability of the element being visited following the efferent direction. These probability values form the basis of the popularity ranks and the significance ranks. Based on the ranks, we can generate potential CC candidates in two ways:

Homogeneous crosscutting – To look for homogeneous CC candidates, we check if the popularity rank of an element is higher compared to its significance rank by a certain confidence threshold. Therefore, an element, despite of its high popularity, is unlikely to be selected as a CC candidate if it is also very complex or aggregates key functionalities of the system by referencing other significant elements.

Heterogeneous crosscutting – To look for heterogeneous CC candidates, we first define a *core set* consisting of elements having much higher significance ranks compared to their popularity ranks. Elements in the core set are unlikely to be CC elements. We then list all elements known to this core set as the candidates of heterogeneous crosscutting. Locating core elements prior to finding heterogeneous crosscutting concerns is an effective manual approach we proposed and verified in [21]. In the manual approach, both core and CC elements are identified through source code reading without exploiting the automated ranking method developed in this paper.

To evaluate the ranking-based approach, we have implemented the Prism Aspect Miner (PAM), leveraging a Java code query engine, the Prism Query Language (PQL) that we previously developed [22]. We have used PAM extensively on various kinds of Java projects ranging from small applications such as Jhotdraw and medium-sized middleware to large-scale systems such as the WebSphere Application Server. Our quantifications show that, on the medium-sized middleware application, PAM is capable of producing hundreds of CC candidates which comprise 70% of what we had manually identified during the course of a few months.

The afore-mentioned random walks resemble that of a random web crawler model in the page-rank algorithm [10]. We modify and extend the page-rank algorithm to the context of aspect mining and make the following contributions in this paper:

1. We first describe the Markov model for computing popularity and significance values of elements in the coupling graphs. This model is inspired by the page-rank algorithm with important modifications to reflect the nature of program sources.
2. We present the detailed design of the PAM aspect mining algorithm including the construction of the coupling graph, the natural and differentiated ranking methods, and different modes for CC selection.
3. We provide a thorough evaluation of the properties of PAM. The evaluation includes the quantification of conventional information retrieval metrics as well as qualitative assessments from domain experts. A runtime profile, characterizing PAM's performance is also presented.

The rest of the paper is organized as follows: Section 2 describes the background information necessary for the understanding of the rest of the paper; Section 3 describes our algorithm in detail; 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 compares our work to the related work.

[1] Jhotdraw is a well-known graphical editing package for investigating crosscutting concerns. URL: `http://www.jhotdraw.org`

2. Background

2.1 Coupling graphs

In this section we define the efferent and the afferent coupling graph, as well as the efferent and the afferent relations for program elements.

We define an efferent coupling graph $G_\epsilon =< V, E >$ where V is a set of program elements: $V = \{c_1, c_2, \cdots, c_n\}$. A directed edge e_{pq} denotes that the element c_p syntactically "knows about" the element c_q. With respect to a specific program element c, the *afferent* relation captures the elements that know about c, and the *efferent* relation captures the elements that are known to c. Suppose we construct a coupling graph as illustrated in Figure 1(A) based on the directions of method invocations among 7 modules. Figure 1(B) and (C) represent the afferent and efferent relations for module 4, respectively. Intuitively, in the efferent coupling graph, crosscutting concerns have high afferent values, and core elements have high efferent values. For the convenience of computation, we define the afferent coupling graph G_α as changing the direction of all the edges in the efferent coupling graph G_ϵ.

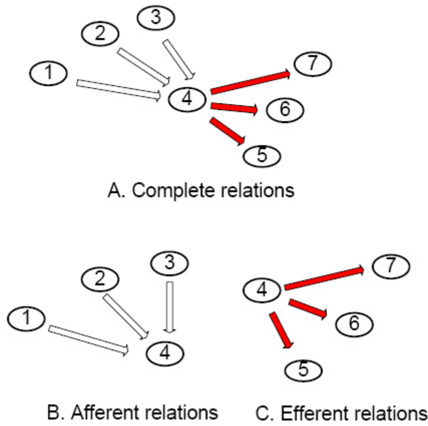

Figure 1. Relations of modules

2.2 The Page-rank algorithm

In this section we review the popular page rank algorithm [10] and provide an interpretation for its use in the context of doing random walks on the coupling graph. A vertex receives higher probability of representing a CC element if it is referenced by a large number of vertices or by vertices with high probability of representing CC elements. Conversely, a vertex is likely to represent a core element if it references a large number of vertices or vertices having good chances to also represent core elements. In the conventional page-rank algorithm, the probability of visiting vertex v_j is expressed formally as:

$$P(v_j) = \sum_{i \neq j}^{n} P(i_to_j) * P(v_i)$$

where $P_(i_to_j)$ is defined as:

$$P_(i_to_j) = \begin{cases} \lambda * 1/Outdegree(v_i) + (1-\lambda) * 1/|V| \\ \quad (if\ Outdegree(v_i) \neq 0) \\ \\ 1/|V|\ (if\ Outdegree(v_i) = 0) \end{cases} \qquad (1)$$

The factor λ is called the *damping factor* for representing a random transition from node i to node j even if $e_{ij} \notin E$. Google uses 0.85 as the damping factor.

The equations above define a Markov process for calculating the probability vector, $\vec{P}(v)$, for all vertices, as the result of the matrix multiplication:

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

where M is the transition matrix defined as: $m_{ji} = P_(i_to_j)$. It is a well-known mathematical fact[2] that the probability vector $\vec{P}(v)$ converges with the repeated application of multiplication. That is, the vector $\vec{P}(v)$ exists such that $\vec{P}(v) = M \times \vec{P}(v)$. In this case, $\vec{P}(v)$ is the eigenvector of M when the eigenvalue of M is 1.

We use the notations $\mathcal{M}_e$ and $\mathcal{M}_a$ to denote the transition matrices for the efferent and the afferent graphs, respectively. In our algorithm, the converged probability vector of all elements are computed for both the efferent coupling graph and the afferent coupling graph. We denote these vectors as the efferent vector: $\vec{\epsilon}$ and the afferent vector: $\vec{\alpha}$. The vectors $\vec{\epsilon}$ and $\vec{\alpha}$ form the basis for generating a total order, or ranks, for all elements reflecting their degrees of popularity and significance.

For ease of presentation in later sections, we define a procedure: `RANDOM_WALK` to encode the operations described above as follows:

`PROCEDURE: RANDOM_WALK(`$\mathcal{M}$`)`
`INPUT:` $\mathcal{M}$, the transition matrix of the relation graph, $\mathcal{G}$
`OUTPUT:` The crosscutting rank, $Rank_{cc}$, or the non-CC rank, $Rank_{core}$.

2.3 Prism query language

Central to PAM is the Prism Query Language (PQL) [22], a declarative query language we developed for Java systems. PQL leverages the AspectJ type patterns with scope operators to simultaneously express searching criteria for Java elements in both their definition patterns and usage patterns. Detailed information and the executable of PQL are publicly available[3]. In Table 1 we give a few examples of the typical PQL statements applied 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. Section 5 presents the detailed quantifications of this property.

3. Algorithm

3.1 Overview

As a duality, the presence of crosscutting concerns systematically increases the afferent complexity of the CC elements and the efferent complexity of the core elements. The ramifications of this kind of systematic effect can be exploited by tools to track the footprints of CCs. Conventional aspect mining analysis primarily focuses on capturing the afferent complexity of CCs through metrics or properties such as "fan-in degree" [9], clones [3], and "degree of scattering"'[19]. We believe that the efferent complexity of core modules are also important to study because they can lead us to measure the "unlikelihood" of a module getting classified as a CC element.

We propose an aspect mining algorithm that exploits both the efferent and the afferent information of a system. We use the random walk algorithm to compute both the popularity ranking and the significance ranking for all nodes in the coupling graphs. Our computation model is derived from the page-rank algorithm [10] to

[2] For more mathematical insights, please refer to *The World's Largest Matrix Computation*: `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/oct02_cleve.html`

[3] Prism Query Language. `http:\\www.eecg.utoronto.ca\~czhang\pql`

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 1. PQL Examples

reflect the nature of program source investigations. In the following sections, we first present a detailed description of this derivation. We then explain how afferent and efferent coupling maps are constructed, followed by the complete description of the algorithm.

3.2 The computation model

We observe some important differences in traversing program sources compared to traversing hyperlinks in web documents. First, the number of elements in a coupling graph of program sources can be significantly smaller compared to billions for web pages. Second, relations between program elements are usually well defined compared to the uncontrolled structure of the links in web pages. Lastly, good ranking decisions have to be made on all elements in the source code, while the computation for ranking web pages favors the ones referred by others. In reaction to these differences, we have made a few modifications to the canonical page-rank algorithm. These modifications are based on our observations and practices. We believe a theoretical analysis of these differences is also possible but defer this to future work.

Small restarting probability

The restarting probability is the probability of the random walker choosing, with equal chances, any other disconnected nodes instead of following an relation link. It is defined using λ in Equation 1 as $1 - \lambda$. λ has a significant impact on the ranking results of the underlying graph. For instance, it has been shown [23] that λ is a useful tool for detecting link-spams in ranking web pages. For program sources, we choose $\lambda = 0.95$, a bigger 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. This counts for the kind of out-of-band dependencies between program elements such as inter-process communications, reflective invocations, or other accidental dependencies that are difficult for the type-based analysis to detect.

Biased transitions

The probability assignment in Equation 1 is an unbiased probability assignment where each node has equal chance of transitioning to a connected node regardless of the number of actual links between two nodes. This idempotent setup is necessary for resisting malicious manipulations of ranking computations in uncontrolled settings such as the Web. As an aspect miner reading the source code, more references to a particular program element will likely bring that element under his attention, hence, increase its chance of being visited. For instance, when examining the `drawFrame` method of the type `TextFigure` in the Jhotdraw application in Figure 2 (Left), an unbiased probability assignment states that the miner is equally likely to visit 5 other class types (underscored in the code snippet) due to either method calls or field accesses. Realistically, the miner 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 introduce the biased transition where we associate each edge e_{ij} with a weight ω_{ij}. 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:

$$P_(i_to_j) = \begin{cases} \omega_{ij} / \sum_{k \neq i}^{n} (\omega_{ik}) & if \;\; e_{ij} \in E \\ 1/|V| & if \;\; e_{ij} \notin E \end{cases} \qquad (2)$$

Treatments of roots

In Equation 1, the rank of root vertices is computed as: $P(v_{root}) = 1/|V| \times \sum P(v_{leaf})$. When $|V|$ is large, root vertices typically receive lower ranks compared to other nodes in the coupling graph. Consequently, on the efferent graph, the root vertices almost always get classified as non-CC elements for receiving smaller efferent rank compared to its afferent value, and the root vertices of the afferent coupling graph are classified as CC elements for the same reason.

It is acceptable for the root vertices of the afferent graph to receive low significance ranks since these vertices represent modules that are simplistic, i.e., not coupled with any other modules in the system. The same conclusion cannot be drawn about the root vertices of the efferent coupling graph. In Figure 2 (Right), we show the Java code for the exception `UnknownException` in a middleware implementation chosen for our experiments. The `UnknownException` is not referenced by any type in the code space, and all other types (underscored) coupled with it receive very high popularity ranks. A human miner would consider this type also as a popular type since its functionality is built purely on popular concerns.

To generalize this observation, we think that the popularity value of a root vertex should be decided by the popularity values of the elements it "knows about". Therefore, we re-compute the efferent probability for a root vertex, v_r, by multiplying the popularity measures, represented by the afferent vector $\vec{\alpha}$, with the $r - th$ column vector of the efferent transition matrix, $\mathcal{M}_e$. The updated $\vec{\alpha}$ is used in producing the final ranks.

Rectifying ranking inversions

Compared to web pages, the elements in program sources are much smaller in number. This is problematic for the treatment of leaf vertices in the original page-rank algorithm. In the original algorithm, the leaf vertices are fixed with artificial transition links to all other nodes in the graph. This can cause inversion of ranks for certain graph structures when $|V|$ is small and $P(v_{leaf})$ is large, i.e., many leaf nodes also receiving high ranks. For illustration, in Figure 3, we depicted a hypothetical call graph of 10 vertices. Vertex 8 and 9 behave "crosscuttingly". But vertex 10 should receive the highest ranking since it crosscuts vertex 8 and 9. Our ranking computation confirms this observation. However, suppose that we start adding new vertices to the call graph and connect every vertex in the original graph with the new vertices to represent the use of certain crosscutting functionalities. When we add a third such vertex, vertex 13, to the call graph, vertex 8 is ranked higher than vertex 10 due to the propagation of $P(v_{leaf})$ along the afore-mentioned artificial links.

Ranking inversion is important to rectify since, from our observations, many classic crosscutting concerns such as logging and exception handling are often leaf vertices in the call graph and, at the same time, used quite frequently in the program. Since these vertices typically receive large ranks, it is important to lower the

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

```java
final public class UnknownException
extends org.omg.CORBA.SystemException
{
    public Throwable originalEx;

    public UnknownException(Throwable ex)
    {
        super("", 0, org.omg.CORBA.CompletionStatus
                .COMPLETED_MAYBE);
        originalEx = ex;
    }
}
```

Figure 2. A. Biased transition (Left) B. Classification of root (Right)

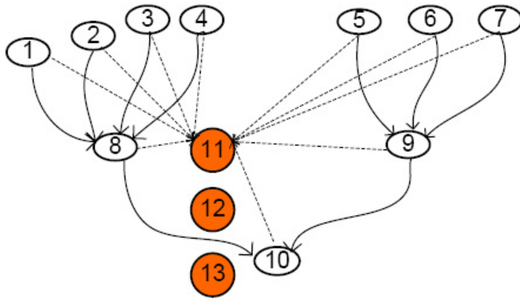

Figure 3. Ranking Inversion. (*The red nodes represents new vertices being added. Node 12 and 13 connect to other nodes in the same way as Node 11*)

possibility of skewing the true ranking result caused by the artificial links established in the canonical page rank algorithm. To minimize ranking inversion, both $\vec{\epsilon}$ and $\vec{\alpha}$ are computed in $|V|$ iterations. Before each iteration starts, the top ranked vertex from the previous iteration is excluded for the ranking computation. The iterations also stop if excluding the previous top-ranked vertex causes the entire call graph to disconnect.

3.3 The construction of the coupling graph

The vertex in the coupling graph (Section 2) can can be mapped to an arbitrary definition of the grouping of program instructions. In our experiments, we map the vertices to collections of the modules defined by the Java programming language including packages, types, and methods. In the coupling graph, $\mathcal{G}$, an edge between vertices A and B represents the following basic relations between the two corresponding program elements:

1. Type extension: A extends B in the type hierarchy. For instance, either A or a type contained in A is a subclass or an interface implementation of either B or a type contained in B.
2. Message sending: A sends a message and the message signature is declared in B. This is to account for polymorphic method invocations. For instance, in the *Observer* design pattern, a call to `addObserver` is not attributed to the specific type implementing the `Subject` interface but to the actual `Subject` interface itself.
3. Reference: A has B as a field or method parameter or A accesses the static members of B.

In addition to basic relations, we also support the qualified relations. Qualification rules can be applied to filter out the basic relations within the same type, the same package, or the same component. The qualification levels can be set as an external parameter. For instance, for software systems consisting of hundreds of components, our graph needs to only capture inter-component relations, e.g., messages sent to types belonging to a different component, if we are to produce component-level rankings. Other kinds of relations are disqualified. Through these relations, we can construct the "knows-about" graph, i.e., the afferent coupling graph, for different levels of source groups such as components, packages, types, and methods. The "known-by" graph, i.e., the efferent coupling graph, can be obtained by changing the direction of all edges in the efferent graph. We use the procedure `MAKE_GRAPH` to generate the transition matrices for the coupling graphs just described.

`PROCEDURE: MAKE_GRAPH(`$\mathcal{C}$`,` $flag$`)`
`INPUT:` (1) $\mathcal{C}$, the collection of concerns being mined; (2) $flag$, a boolean value
`OUTPUT:` The efferent transition matrix, $\mathcal{M}_\epsilon$, if $flag = true$; the afferent transition matrix, $\mathcal{M}_\alpha$, if $flag = false$.

3.4 Selection of CC candidates

In our algorithm, the nature of a program element can be determined with a straightforward comparison between an element's popularity rank and its significance rank. The implementation of this general classification rule requires us to address two technicalities: 1. what is considered more significant or popular? 2. what is the threshold used in the comparison for making the classification?

The computation of ranks
CC candidates are selected based on both the efferent and the afferent ranks computed from the coupling graphs. The ranks of program elements can be either mapped directly to the natural order of the numerical values in vectors $\vec{\epsilon}$ and $\vec{\alpha}$ or based on the differences between their natural orders in these two vectors. We refer to the first type of ranks as *natural ranks* and the latter *differentiated ranks*. The *natural ranks* reflect the physical uses of elements in the program sources, and *differentiated ranks* favor large differences between the values in the vectors $\vec{\epsilon}$ and $\vec{\alpha}$. For instance, a popular element can receive a high rank in the natural popularity rank but not necessarily so in the differentiated popularity rank if its significance rank is also high. The selection between these two ranking methods is defined as an external parameter, `TYPE`, to our previous definition of the procedure `RANDOM_WALK`.

Confidence level
We use the term *confidence level* l to denote the threshold for the difference between afferent ranks and efferent ranks for the algorithm to make a classification decision. It is defined as the ratio between an element's popularity rank and its significance rank. Confidence levels can be used to control the number of candidates output by the algorithm and is left as a parameter of the algorithm.

```
PROCEDURE: I_SELECTION
INPUT: Rank_ε, Rank_α
OUTPUT: A set, S_hetero, of heterogeneous CC candidates, computed in the following algorithm.
IMPLEMENTATION:
  Let S = {c|Rank_ε(c)/Rank_α(c) > l} { Generate the non-crosscutting set as described previously.}
  S = GETCHILDREN(S) { GETCHILDREN is a trivial procedure for retrieving the subtypes of types in S}
  Let S' = ∅
  while |S'| < n do
    if e_jk ∈ E and c_j ∈ S and c_i ∉ S then
      S' = S' ∪ c_k {Include concern c_k if it is not in S but referenced by a concern in S}
    end if
  end while
  Return S'
```

Figure 4. Indirect selection

Modes of selection

Based on the afferent ranks ($Rank_\alpha$) and the efferent ranks ($Rank_\epsilon$), CC candidates can be selected in three modes: the direct mode, focusing on homogeneous crosscutting, the indirect method, focusing on heterogeneous crosscutting, and the hybrid mode, combining the results of the previous two modes. The modes are defined as an input parameter of the algorithm.

In the *direct selection* mode, we classify element c as a CC candidate if $Rank_\alpha(c)/Rank_\epsilon(c) > l$. Since the direct selection is decided based on an element's popularity in the coupling graph, it favors elements that crosscut the system homogeneously. This action is encoded in the procedure: `D_SELECTION`.

In the *indirect selection* mode, we classify element c as a CC candidate in two steps: we first sort all elements on the ratio $r_c = Rank_\epsilon(c)/Rank_\alpha(c)$; we form the set of the non-CC elements by sequentially selecting element c starting from the highest value of r_c in the sorted set, if $r_c > l$; we then output a set of filtered elements known to the non-CC element set. A heuristic is applied as we restrict the number of non-CC candidates to be at most half of the total number of ranked elements. The design of *indirect selection* follows the general insight from our previous experience [21] that heterogeneous crosscutting concerns are relative to the core ones. This action is encoded in the procedure: `I_SELECTION`(n) in Figure 4. In the *hybrid selection* mode, we output a mixture of candidates employing both kinds of selection methods. Both direct and indirect modes are active for the hybrid selections, and we return an aggregation of results from these two modes. The procedure `HYBRID` is defined as:
`D_SELECTION` ∪ `I_SELECTION`.

3.5 The Prism aspect mining algorithm

Having introduced the fundamental elements of the mining algorithm, we now present the complete procedure for finding CC candidates. To improve the quality of the ranking, we first introduce the procedure `PARTITION` for separating modules that are unrelated in the type space so that elements in the same ranking are truly relevant to each other. This procedure is implemented by a simple recursive traversal of the original graph, we omit the details here.

`PROCEDURE: PARTITION`($\mathcal{G}$, $threshold$)
`INPUT:` (1) $\mathcal{G}$, the coupling graph constructed for all concerns in the system; (2) $threshold$, the degree of knowledge between two concerns for them to be considered unrelated. For instance, in our current implementation, this value represents the number of class types known across either components or packages, and it is set to '1'.
`OUTPUT:` A set, $P = \{P_1, P_2, \cdots, P_n\}$, of n subgraphs of $\mathcal{G}$.

```
PROCEDURE: MINING(source, confidencelevel(l), mode)
INPUT: (1) source, the source code of the target system; (2) l, a float denoting the confidence threshold; (3) mode, one of three modes: direct, indirect, and hybrid.
OUTPUT: A set, CC, of concerns selected as crosscutting candidates for each partition.
IMPLEMENTATION:
  G = INITGRAPH(source)
  P = PARTITION(G, 1)
  for each P_k ∈ P do
    Rank_α = RANDOM_WALK(MAKE_GRAPH(P_k, true), mode)
    Rank_ε = RANDOM_WALK(MAKE_GRAPH(P_k, false),mode)
    if mode = DIRECT or INDIRECT or HYBRID then
      CC = DIRECT or INDIRECT or HYBRID (Rank_α, Rank_ε, l)
    end if
    CC = GETCHILDREN(CC)
  end for
```

Figure 5. The Prism Mining Algorithm

For completeness, we introduce a trivial procedure: `INITGRAPH`($source$) that returns $\mathcal{G}$, the initial coupling graph for all concerns of the system through the PQL query engine. The complete algorithm is presented in Figure 5:

3.6 Example

We now use a simple example to illustrate the afore-introduced mining process. Our example[4] is an adapted version of the "telecom" illustration in the source distribution of the `AspectJ` compiler. "Billing" and "Timing" are two features originally written as aspects. In the adapted version, we inject these two features in plain Java so they become the target of the mining effort. At the same time, we inject two additional features representing two different kinds of crosscutting. For the **Logging** feature, we converted the original "print" statements to calls to a `Logger` object. The `Logger` class uses the `StorableOutput` to write persistent logs. This is a typical case of a homogeneous crosscutting concern. For the **Persistence** feature, we injected persistence capabilities into classes `Call`, `Connection`, and `Customer` by having them implement the `Storable` interface. This is also a typical homogeneous crosscutting concern in the type space.

Figure 6 (A) gives the UML diagram of our simple telecom application. Figure 6 (B) shows the coupling graph after superimposing the type relationships onto the invocation map. Without much analysis, the CC candidates in our simple example should ideally include `Billing`, `Timer`, `Storable`, `Logger`, and `StorableOutput`. `StorableOutput` is also a CC type because it is part of the "logging" functionality in spite of being only used exclusively by `Logger`. A contrary scenario is that the types such as `Customer` and `Connection` have high degree of fan-ins. However, they carry out the basic logic of the system, hence, cannot be classified as CC candidates. We illustrate how these scenarios are correctly treated in our ranking analysis.

Table 2 lists the actual probability values and the respective rankings (the larger the rank value the higher the rank) for all the types in both the afferent and the efferent coupling graph. We can observe that the algorithm ranks the type `StorableOutput` as the

[4] This example is publicly available at `http://www.eecg.utoronto.ca/~czhang/mining/tele.zip`

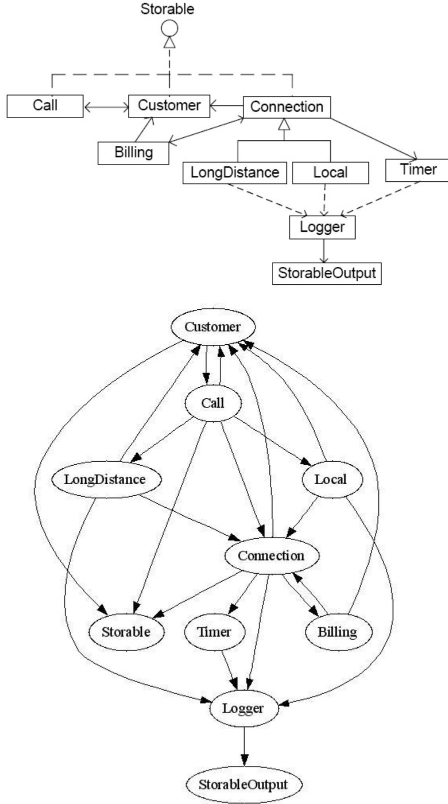

Figure 6. A: UML class diagram. B: Concern graph

highest crosscutting type even though it is only known by one other type. The main types: `Call`, `Connection`, and `Customer`, as expected, have high crosscutting rankings. However, they have even higher afferent rankings which prevent them from being considered as CC candidate. For illustration, we generate the candidate set using the differentiated ranks and the hybrid selection mode with a confidence level of 1. That is, our classification decision is simply based on which of the two ranks for a particular type has a larger value. We use unbiased walks due to the small size of our example.

The `D_SELECT` procedure picks the types `StorableOutput`, `Logger`, `Timer`, and `Storable` since they have higher afferent ranks. The `I_SELECT` procedure first picks the types `Call`, `Customer`, `Connection`, `LongDistance`, and `Local` as non-CC candidates. The type `Billing` is not selected due to our capacity control heuristic mentioned previously. By following outgoing edges from the non-CC candidates, the `I_SELECT` procedure generates a CC-candidate set consisting of `Timer`, `Billing`, and `Logger`. We then have our final result set consisting of the desired CC candidates.

4. Implementation

4.1 The Prism aspect miner

The Prism aspect mining algorithm is implemented in Java as the Prism Aspect Miner (PAM) and publicly available[5]. PAM requires the Prism query language package, – PQL, which provides both the compiler extension to AspectJ for the extraction of coupling

Type	Prob	Rank	Prob	Rank
	Afferent		Efferent	
Local	0.05227	1	0.0943	6
LongDistance	0.05227	2	0.0943	7
Billing	0.0550	3	0.0920	5
Timer	0.0550	4	0.0073	3
Call	0.09690	5	0.2727	10
Connection	0.1115	6	0.1537	8
Customer	0.1327	7	0.2657	9
Storable	0.1365	8	0.0050	2
Logger	0.1404	9	0.0098	4
StorableOutput	0.1673	10	0.0050	1

Table 2. Ranking scores

graphs and the underlying querying capabilities. A large number of commandline options are available for setting various parameters of the algorithms presented in Section 3. These options include confidence levels, selection modes, qualification levels, types of ranks, and others. The detailed listing of options and instructions of how to use them are published online.

4.2 Domain knowledge injection

PAM also provides a few options that allow the users to influence the default behaviors of PAM by injecting the domain knowledge about the investigated system. Domain knowledge injections are in the forms of declarative PQL queries and can leverage the full descriptive capabilities of PQL. PAM 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 5.6). The option `ignore`, taking a PQL query as its value, excludes non-interested program elements for a particular run of PAM. For example, the query `match type:"org..samples.*"` filters out all sample code shipped with JHotdraw version 6.
2. **Specialization** – Similar 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 PAM 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 PAM 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 also 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. PAM is capable of provisioning these three kinds of *user-defined concepts* by reading a user-defined concept file through the `group` commandline option. The file contains key-value pairs associating an unique concept name with a PQL

[5] The Prism Aspect Miner. `http://www.eecg.utoronto.ca/~czhang/mining`

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 3. Effects of using customized concept definitions

query. This name, representing the user-defined set, is used by PAM 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.

In Table 3, we compare the top-15 ranked elements computed using plain Java types (2nd column) to using concept customizations (3rd column). The reason for using the customized concepts is based on the observation that the types representing the features "persistence" and "undo" are repeatedly reported in the plain ranks. For PAM 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 in a property file as follows:[6]:

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

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

5. Evaluation

Aspect mining algorithms can be evaluated effectively if there exist commonly recognized benchmarks for measuring the quality of the mining results. Unfortunately, such benchmarks are not known to us. We therefore leverage, for the bulk of the evaluation presented in this paper, our extensive domain knowledge about CCs in middleware implementations [19, 21]. We use the typical information retrieval metrics *precision* and *recall* to measure the quality of our algorithm against results obtained from manual mining efforts. We also offer a qualitative validation of our mining results, obtained with PAM on JHotdraw, against the mining results reported in the literature. We then report on mining experiments using the AspectJ compiler and IBM's WebSphere application server (WAS) as mining target. We also report on the runtime characteristics of PAM.

5.1 Quantifying the mining of ORBacus

ORBacus is IONA's CORBA[7] product. It is distributed as Java sources. We have thoroughly studied the crosscutting concerns in the architecture of ORBacus [19] and refactored a large number of them. The remnants of the removal of CCs from ORBacus consist of 855 classes (i.e., the core functionality of ORBacus.) There are about 960 class types belonging to crosscutting concerns in the full ORBacus distribution, not counting newly created aspect modules. We use these two groups of types as the reference data sets, and our experiment studies the effect of the algorithm's confidence levels, selection modes, and types of ranks on the measures of precision and recall. This experiments serve to calibrate the algorithm for latter experiments, where no reference sets are available. However, note that the precision and recall in our experiments can not be interpreted as the absolute quality measures of the algorithm since our manual classification is subjective and partial. There exist additional previously unknown CCs. We use these measures to reflect how close the automated mining is to our manual effort. It is a relative measure.

Figure 7 shows the relationship between confidence levels and the precision and recall values for all three modes of CC selections labeled as: D (Direct), I (Indirect), and H (Hybrid). The top chart reports measures for the *natural ranking*, and the bottom for the *differentiated ranking*. The X-axis represents 20 incremental steps of confidence levels: the first 10 points represent the micro-intervals in increments of 0.1 between 1 and 2, and the last 10 points represent the interval in increments of 1 between 2 and 11. The highest precision of both ranking methods are between 50% to 60%, and the recall measures are between 20-30% in the micro-interval for both methods. The *direct* selection produces higher precision using the natural ranking but generates more false positives using the differentiated ranking. The hybrid mode shows the reconciliation effect for the direct and the indirect modes in the precision measure. It has the highest recall rate due to combining the results of both modes. The recall rates decline as l increases. This effect is even more pronounced for the natural rankings.

We have noticed some problems with our initial applications of PAM (see Figure 7.) The accuracies of the classification is not sufficiently high and the recall rate is generally low. A closer investigation reveals that the classification errors are of the following nature:

1. *Support code*: The ORBacus sources include a large amount of IDL compiler-generated support code. This code is not part of the functional implementations of ORBacus itself. These types include "stubs", "helpers", and "holders" comprising 905 classes out of a total of 1815 classes in ORBacus. The large pieces of support code can seriously promote the CC rankings of the core functionalities due to their widely scattered uses in the support code. To better assess the ranking qualities of PAM, the support code should be excluded from the reference sets.
2. *Interface types*: 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 [15]. However,

[6] The full options used for generating the rankings in this example are: `-xbinary all -qualify package -ignore "match type:\"org..samples..*\";" -group jhotdraw.properties -ranktype natural -confidence 1`

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

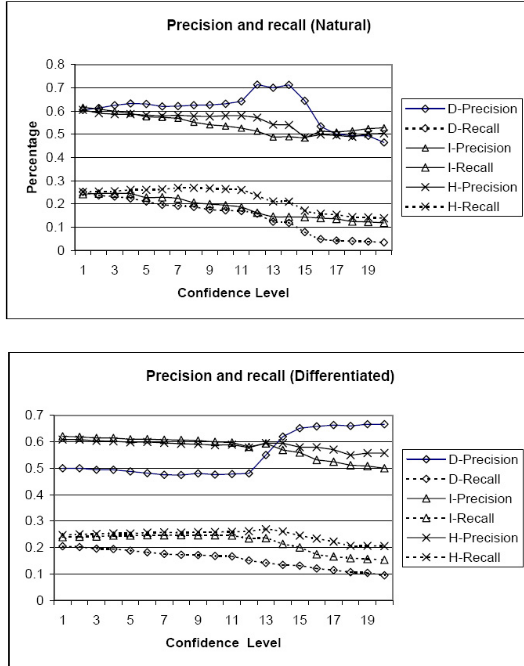

Figure 7. Precision and recall (initial)

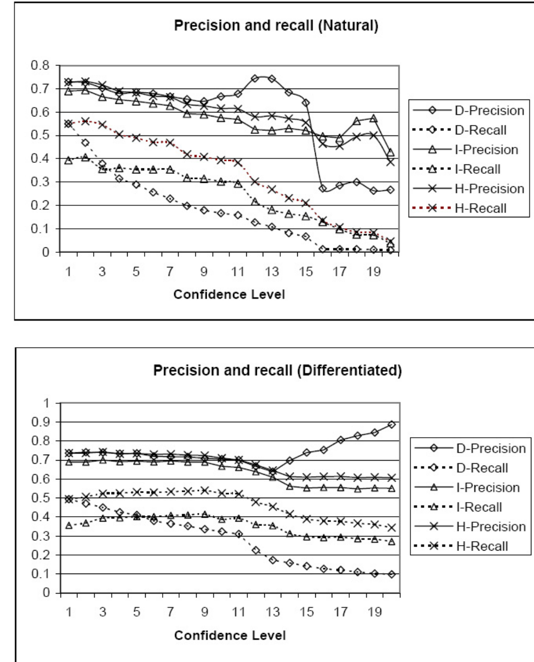

Figure 8. Precision and recall (fixed)

we have not exercised this treatment in our previous manual identifications. For the improved evaluation of PAM, we move the classes of which the names end with "Operations" from the non-CC reference set to the CC reference set.

The new reference data sets contain 480 CC classes and 332 non-CC classes. The precision and recall measures for the new reference data is shown in Figure 8. For the more accurate reference data, we observe significant improvements in both precision and recall measures. The highest precision is 88.7% with 10% recall using the direct selection mode when $l = 20$. The highest recall is 54% with a precision of 72% using the hybrid mode at $l = 1.9$. We believe this level of accuracy is capable of giving good results compared to our manual method.

From these experiments, we draw two conclusions: 1. The hybrid mode is generally more stable as l varies, and it also has good precision and the highest recall values; 2. the differentiated ranking is less sensitive to the changes of l compared to the natural ranking in the micro interval. Therefore, the default mode of PAM is set to use the hybrid mode and the differentiated ranking.

5.2 The mining of JHotdraw

JHotdraw[8], since its original adoption for the illustration of crosscutting concerns, has been an application for many aspect mining studies [9, 16, 14, 20]. It is not yet known if there exists a commonly accepted list of CCs in JHotdraw. To compare and validate our mining results on JHotdraw, we collect CCs discovered by previous efforts and examine whether we can produce the same results.

PAM produces 95 crosscutting candidates for JHotdraw version 6 which contains 279 classes using a confidence value of 1.9 with the hybrid mode[9]. From the afore-mentioned prior research on JHotdraw, the following types are considered as crosscutting concerns: figure selections (`FigureSelection`, `FigureSelectionListener`), persistence (`Storable, StorableOutput, StorableInput`), and undo `Undoable, UndoManager`. These classes are ranked as 25th, 30th, 2nd, 5th, 6th, 35th,and 36, respectively. Thus, PAM confirms existing results and locates other additional aspect candidates.

5.3 The mining of the AspectJ compiler

The AspectJ compiler, largely implemented in Java, could inherently incur a certain degree of crosscutting. We have used PAM to mine the source code of the AspectJ 5.0 compiler, consisting of close to 1000 classes. We did not include a small portion of code written in Java 5 and the entire AspectJ Eclipse plug-in (AJDT). The PAM parameters are the same as in the previous experiment, and the mining result is publicly available[10]. To assess the quality of the mining results, we consulted with the AspectJ developers. Our first attempt processes every inter-class relation. This led to a large number of top-ranked elements to be 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 relations if both classes are not defined within the same package. We confirmed with the domain expert that the quality of the ranks has improved. 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 PAM:

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.

[8] JHotdraw. `http:\\www.jhotdraw.org`

[9] The detailed JHotdraw result is published at :`http://www.eecg.utoronto.ca/~czhang/mining/j6.txt`

[10] AspectJ mining results. URL:`http://www.eecg.utoronto.ca/~czhang/mining/aj.txt`

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*"[11] when unanticipated events occur during the compilation and the weaving process.

5.4 Large scale aspect mining

System software such as middleware is an active area where aspect orientation is being experimented with to make architectural improvements. Colyer and Clement [4] have shown that aspect oriented development can be carried out even on very large scale middleware systems such as IBM's Websphere Application Server. For such large systems, aspect mining is especially beneficial for gaining insights about crosscutting concerns that are hard to obtain manually. We use PAM to mine the source code of the Websphere Application Server with two objectives: 1. To confirm the results of the manual investigations in [4]; 2. to reveal new insights about the crosscutting phenomena of the software components within the Websphere Application Server. These mining results could serve as key directives for the continued product-line engineering of the Websphere Application Server through aspect oriented refactoring.

Our mining target is the 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. The partitioning procedure divides 125 components of the Websphere Application Server bulid into 36 unrelated partitions. And one of the partitions contains 89 connected components and, hence, is the focus of our discussion.

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. The functional implementations of these two components (ranked 4th and 8th in non-CC ranking), which provides the logic for the interfaces, are not categorized as CC candidates. In addition, we selectively discuss some of the new CC concerns for the Websphere Application Server architecture:

1. Object Request Broker: the WebSphere IPC component implements the CORBA interfaces which are heavily used for inter-process communications within WebSphere internal components [12]. This is an example of the object distribution concern crosscutting with the operational logic of individual components.
2. National Language Support (NLS): the component `nls` carries the responsibility of translating generic Websphere Application Server messages to appropriate messages in the configured encoding of the system. This is a pervasive concern spanning over 10 components.
3. Java Messaging Server: the Websphere Java messaging component, only directly referenced by a few other components, can be very easily classified as having little crosscutting impact. Its high ranking on the CC-rank is due to the subtypes of its types defined in both the WebSphere component framework and serviceability functionality. These two functionalities are widely used in the Websphere Application Server architecture, hence, contribute significantly to the rank of the Java messaging component via the transitivity property of the random walk algorithm.

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. These specific consoles include distribution management, environment configuration, performance tuning, and others. 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" [12]. 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.

5.5 Runtime characteristics

To quantify the efficiency of PAM, we have chosen 7 Java applications of various types and sizes: graphical editing (i.e., JHotdraw), databases (Prevayler[12], hSQL[13], Derby[14]), middleware implementations (ORBacus, Websphere Application Server, and PADRES[15]) and the AspectJ compiler version 5[16].

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 PAM to generate the mining results. The PQL indexer incurs negligible

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

[12] Prevayler. URL: `http://www.prevayler.org`

[13] HSQL Database Engine. URL: `http://www.hsqldb.org/`

[14] Apache Derby. URL:`http://db.apache.org/derby/`

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

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

	No. of Types	Duration(sec)	LOC
Prevayler	38	0.15	2396
Padres	203	1.04	17124
HSQL	310	1.77	48300
JHotdraw6	398	1.59	15541
Derby	1261	15.9	153000
aj1.5	1353	11.67	89634
ORBacus	1815	24.1	64704
WAS	8800	1085	2000000

Table 4. Mining performance

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 PAM is set at 512M for all experiments except for the WebSphere application server, where it is set to be 1.5G. Table 4 shows that for most mid-sized applications, PAM is able to produce the results in less than 30 seconds. Scaling up to large scale applications such as the Websphere Application Server, PAM requires 18 minutes to complete on our workstation (a conventional desktop PC[17].)

5.6 Lessons learned

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

1. ***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. It is not difficult to filter out these false positives with the domain knowledge of the application at hand.
2. ***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 context-sensitive ranking capability of PAM can be leveraged to achieve this.
3. ***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 manually classify 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.

In addition to misclassifications, 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.

6. Related work

Research in the area of aspect mining and CC discovery can be roughly classified into three categories: static analysis, runtime 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 his concern discovery-by-query task over large code bases. AMT [5] and AMTex [18] enable the specification of crosscutting concerns using both type and lexical patterns. JQuery [7], CME[18], and PQL [22] provide language-based approaches to improve the expressiveness of this specification. FEAT [11] is based on recording the code browsing and code manipulation process to track and map concerns. All these approaches are more manual, query-based, and assistive 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 [9] carried out a fan-in analysis on various systems to account for the "popularity" of crosscutting types. Bruntink *et al.* [3] presented the detecting of scattered code clones for locating crosscutting concerns. Our earlier work has introduced the notion of "degree of scattering" [19] to produce ranks of frequently used types and methods in Java systems. Compared to this class of approaches, our random walk based algorithm presented in this paper is able to reflect the transitive nature of the "popularity" of program elements. Our simultaneous use of "popularity" and "significance" 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.* [14] demonstrated the effectiveness of using formal concept analysis over execution traces of a program. Breu *et al.* [1] 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.

Muli-model analysis means incorporating artifacts other than the program source for the purpose of locating crosscutting concerns. Shepherd *et al.* [13] leverage natural language processing capabilities together with the keywords and comments of the source for clues about crosscutting concerns. Breu and Zimmermann [2] have developed a scheme for exploiting CVS histories in tracking crosscutting updates. Yu and Mylopoulos [17] have shown that certain structures in the goal model underlying the requirements of an application can lead to the discovery of crosscutting concerns. These approaches complement existing mining approaches.

Inoue *et al.* [6] presented an application of the page-rank algorithm for ranking components in Java program sources. The ranks,

[17] 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.

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

interpreted as weights, are equivalent to our popularity ranks used to identify "fundamental and standard" [6] types. Aside from solving a different problem, – the problem of CC discovery,– our algorithm has many significant technical differences. For example, we adjust the page-rank 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.

7. Conclusion

We have proposed the use of random walks to approximate the process of how a human aspect miner distinguishes between core elements and crosscutting concerns without knowing about the application semantics. We first construct both the efferent and the afferent coupling graph for elements from the program sources. The random walk on the efferent graph determines the degree of "popularity" for each element, and the walk on the afferent graph computes the degree of "significance". Our computation model for obtaining the numerical values of "popularity" and "significance" derives from the page-rank algorithm with important modifications and extensions to readily apply it to the context of program source analysis. Based on these two kinds of numerical values, we generate the "popularity" ranks for all concerned program elements as indicators of their "scatteredness", and we generate the "significance" ranks to reflect the degree of their syntactic complexity. These two rankings are used in concert to make the classification decision.

This random-walk based aspect mining algorithm is implemented in the Prism Aspect Miner (PAM) and publicly available for evaluation. PAM automatically partitions the type space to increase the mutual relevance of the ranked elements. Leveraging the flexibility and the efficiency of the Prism Query Language, users of PAM can also produce context-sensitive ranks for selected program elements only. We have evaluated PAM extensively, starting out and calibrating the algorithm, on a legacy middleware implementation that we have previously manually refactored using aspects. Our quantifications show the relationships between the effectiveness of PAM and different parameters in identifying the final CC candidates. We also conducted comparative studies between our mining approaches and results from existing research efforts. Moreover, in evaluating our results, we consulted with domain experts to provide qualitative assessments on applications that aspect mining has never been attempted on. We show the capability of PAM in mining very large-scale code bases such as IBM's WebSphere Application Server. The runtime study illustrates that mining applications can be done very efficiently.

In our future work, we first seek more ways of allowing PAM users to inject domain-specific knowledge into the mining process. Domain specific knowledge can be used to influence all aspects of the algorithm including the graph construction and the parameters in the random walk model such as the damping factor and the transition probability assignments. We also want to integrate PAM with our Eclipse[19]-based aspect-oriented refactoring framework CRAFT[20].

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.

IBM, AIX, DB2, DB2 Universal Database, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. UNIX is a registered trademark of The Open Group in the United States and other countries. Other company, product, and service names may be trademarks or service marks of others.

This research has been supported in part by an NSERC grant and in part by an IBM CAS fellowship for the first author. The authors are very grateful for this support.

References

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

[2] Silvia Breu and Thomas Zimmermann. Mining aspects from history. In Sebastian Uchitel and Steve Easterbrook, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. ACM Press, September 2006.

[3] 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*, 31(10):804–818, 2005.

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

[5] Jan Hannemann and Gregor Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. In *Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, 2001. URL: http://www.cs.ubc.ca/~jan/amt/.

[6] Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Component rank: relative significance rank for software component search. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 14–24, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM Press.

[8] Uira Kulesza, Claudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *9th International Conference on Software Reuse (ICSM'06)*, Philadelphia, USA, 2006.

[9] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 132–141, Washington, DC, USA, 2004. IEEE Computer Society.

[10] 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.

[11] Martin P. Robillard and Gail C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *International Conference on Software Engineering*, Orlando, Florida,

[19] Eclipse. URL:http://www.eclipse.org

[20] CRAFT. URL:http://www.eecg.utoronto.ca/~czhang/craft

USA, May 19-25, 2002 2002.

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

[13] David Shepherd, Lori Pollock, and Tom Tourwé. Using language clues to discover crosscutting concerns. In *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[14] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 112–121, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Paolo Tonella and Mariano Ceccato. Refactoring the aspectizable interfaces: An empirical assessment. *IEEE Transactions on Software Engineering*, 31(10):819–832, 2005.

[16] A. van Deursen, M. Marin, and L. Moonen. Ajhotdraw: A showcase for refactoring to aspects. In *Workshop on Linking Aspects and Evolution (LATE05). 4th International Conference on Aspect-Oriented Programming*, 2005.

[17] Yijun Yu, Julio Cesar Sampaio do Prado Leite, and John Mylopoulos. From goals to aspects: Discovering aspects from requirements goal models. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International (RE'04)*, pages 38–47. IEEE Computer Society, 2004.

[18] 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.

[19] Charles Zhang and Hans-Arno Jacobsen. Refactoring Middleware with Aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1058–1073, November 2003.

[20] Charles Zhang and Hans-Arno Jacobsen. Prism is research in aspect mining. In *Companion of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2004.

[21] Charles Zhang and Hans-Arno Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proceedings of the 19th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, September 2004.

[22] Charles Zhang and Hans-Arno Jacobsen. Prism Query Language:A Crosscutting Concern Investigation Language. Software Demonstration, in 5th International Conference of Aspect Oriented Software Development, March 2005.

[23] 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*, volume 3233 of *Lecture Notes in Computer Science*, 2004.