Identifying Crosscutting Concerns Using Historical Code Changes

Bram Adams Zhen Ming Jiang Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
{bram, zmjiang, ahmed}@cs.queensu.ca

ABSTRACT

Detailed knowledge about implemented concerns in the source code is crucial for the cost-effective maintenance and successful evolution of large systems. Concern mining techniques can automatically suggest sets of related code fragments that likely contribute to the implementation of a concern. However, developers must then spend considerable time understanding and expanding these concern seeds to obtain the full concern implementation. We propose a new mining technique (COMMIT) that reduces this manual effort. COMMIT addresses three major shortcomings of current concern mining techniques: 1) their inability to merge seeds with small variations, 2) their tendency to ignore important facets of concerns, and 3) their lack of information about the relations between seeds. A comparative case study on two large open source C systems (PostgreSQL and NetBSD) shows that COMMIT recovers up to 87.5% more unique concerns than two leading concern mining techniques, and that the three techniques complement each other.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Concerns, Mining Software Repositories*

General Terms

Algorithms, Documentation, Experimentation

Keywords

concern mining, mining software repositories, empirical research

1. INTRODUCTION

A concern is commonly defined as anything that stakeholders consider as a conceptual unit [31]. Concerns range from development-oriented *tracing*, and more general-purpose *caching*, to domain-specific *business rules*. Developers and architects continuously need up-to-date knowledge about concerns currently implemented in their systems, and about the location of these concerns throughout the code. For example, during maintenance and re-engineering, developers need to locate specific concerns in the source code. Bug fixes must be propagated to the whole implementation of a concern [14], and possibly to other concerns with which the concern interacts. Architects need to map the currently implemented concerns to the reference architecture to verify architecture conformance. As such, concern mining is indispensable for software maintenance, reverse-engineering, re-engineering and even for redocumentation.

As manual concern mining is tedious and subjective [1], concern mining techniques support the identification of concerns in software systems. Static techniques analyze source code, dynamic techniques analyze execution traces, and history-based techniques analyze changes in the source code repository. The techniques generate concern "seeds", which are sets of program entities that possibly contribute to the implementation of a concern. In a typical concern mining process [26, 28], developers perform the four steps:

1. generate concern seeds using concern mining techniques;
2. determine whether a seed represents a concern by analyzing how its fragments work together;
3. complete the seed manually if it represents a concern;
4. try to understand the interaction between concerns.

Except for step 1, the concern mining process is purely manual. Concern browsing tools [23, 31] can support the process, but they still depend on the quality of the concern mining results and human input. Due to shortcomings in current concern mining techniques [26], the mining process in practice requires considerable time and resources, slowing down developers. We identified three major shortcomings of current concern mining techniques, which hinder multiple steps of the concern mining process:

S1: Inability to merge seeds with variations

Concerns are often implemented by cloned code fragments [7]. Customization and developer mistakes [6] lead to slight variations between the clones. Concern mining techniques can no longer unify them. This leads to duplicate seeds which developers must process in step 2 of the mining process.

S2: Tendency to ignore important facets of seeds

Most of the mining techniques describe the *behavior* of seeds (functions), but ignore the *state* of seeds (variables and types), and *preprocessor* entities (macros). Hence, mined seeds are less representative of the actual implementation of a concern, making steps 2 and 3 more time-consuming for developers.

S3: Lack of information about seed relations

Composite concerns are large, widely scattered concerns that are hard to maintain. They are usually composed of multiple sub-concerns [10, 11]. Currently, developers need to discover each collaborating sub-concern manually in step 4.

ICSE '10, May 2-8 2010, Cape Town, South Africa

We propose a concern mining technique named COMMIT ("COncern Mining using Mutual Information over Time") that addresses these three shortcomings. COMMIT analyzes the source code history to statistically cluster functions, variables, types and macros that have been changed together *intentionally*. The links between the clusters reveal the relations between seeds. We compare COMMIT to two other state-of-the-art mining techniques [5, 39] on the open source PostgreSQL database and NetBSD operating system. We find that COMMIT complements the other two techniques, while recovering a larger number of unique and rich concern seeds, many of which represent widely scattered, composite concerns.

Our main contributions are as follows:

- Identification and discussion of three shortcomings that hinder the concern mining process.
- Design of a history-based concern mining technique (COMMIT) to address these three critical shortcomings.
- A large-scale empirical comparison of COMMIT with two leading mining techniques [5, 25].

Organization of the Paper. Section 2 introduces relevant concern mining terminology. Section 3 discusses three identified shortcomings of state-of-the-art techniques. Section 4 presents COMMIT, and explains how it addresses the three shortcomings. Section 5 compares COMMIT to a leading static technique, CBFA [39], and history-based technique, HAM [5], by applying these techniques on the PostgreSQL database and NetBSD operating systems. Section 6 discusses threats to validity. Related work is presented in Section 7, and Section 8 summarizes our findings.

2. BACKGROUND

This section introduces important concern mining concepts. For the reader's convenience, we briefly explain how the state-of-the-art CBFA [39] and HAM [5] concern mining techniques work, as we use them throughout this paper to motivate the shortcomings of current concern mining techniques and we compare our COMMIT mining technique to them. We illustrate CBFA and HAM on Figure 1, which contains four successive versions of a simple C system consisting of a `client`, server `front_end` and `back_end`.

2.1 Terminology

Concerns can be categorized based on the modularity of their implementation. **Modular concerns** are the easiest to locate and understand as their implementation is concentrated in one source module or component. **Crosscutting concerns** (CCCs) [24] are concerns that are not modular. Their implementation is spread ("scattered") across multiple modules. Each scattered code fragment of a CCC is called an "instance" of the CCC. The instances of a CCC can be more or less identical to each other ("homogeneous CCC"), or exhibit large variations ("heterogeneous CCC") [13].

A **composite concern** is a CCC consisting of multiple smaller sub-concerns that work together. These sub-concerns are typically spread out across a large number of components and development teams [10, 11]. For example, as encryption support without decryption support is meaningless, a "secure communication" concern consists of multiple instances of an encryption sub-concern (one for each sender) *and* multiple instances of a decryption sub-concern (one for each receiver) that work together.

Concern mining techniques generate **concern seeds**, i.e., sets of related code entities that likely contribute to the implementation (set of instances) of a concern. These techniques focus especially on CCCs, as modular concerns can be easily identified manually. Depending on the intended usage of a technique, it can be applied as frequently as once per release (for documentation) up until once per feature request or even bug report.

2.2 CBFA

Clustering-Based Fan-in Analysis (CBFA) [39] is a recent generalization of the widely used Fan-in static mining technique [25]. CBFA uses the number of unique callers of each function (i.e., Fan-in value) as an indicator of the scattering of the use-sites of that function across the system. CBFA then filters out functions that are invoked too frequently (utility functions and getter or setter functions) or not frequently enough. Functions with sufficiently similar substrings in their names are clustered together to create larger seeds. Finally, CBFA ranks the seeds based on a "cluster Fan-in" metric, i.e., the sum of the Fan-in values of all functions in a seed.

The second column of Table 1 shows the concern seeds identified by CBFA in Version 3 of Figure 1. These seeds are ordered by their cluster Fan-in, then by their **dimension** D (number of program entities that they contain). The `*_log`/`*_lock3` functions rank very high, as they are invoked eight times in total.

2.3 HAM

History-based Aspect Mining (HAM) is a history-based concern mining technique [5]. HAM clusters all functions N that add or remove a call to the same function during a particular time interval or in all change transactions of a particular developer. Then, HAM assigns all functions F that are called by the same cluster of calling functions N to the same concern seed.

The third column of Table 1 shows the mining results of HAM based on the four versions of Figure 1. HAM orders its results first on the seed dimension D, then on the **scattering value** S (number of unique calling functions over which the concern is scattered) [5]. The top results correspond to seeds of up to three program entities, which are called from one or two functions. The logging concern is not reported as a top result since the dimension D for seeds five and six, which correspond to the logging concern, is only one.

3. THREE SHORTCOMINGS

Modern concern mining techniques suffer from three shortcomings that hinder the concern mining process. We discuss these shortcomings by comparing the results of CBFA and HAM on Figure 1 against the desired mining results in the fourth column of Table 1. For each shortcoming, we discuss its motivation, give examples and explain how an ideal mining technique should deal with it. The first shortcoming has been discussed before [22, 26].

S1: Inability to merge seeds with variations

Motivation Concern mining techniques cannot cope well with variations in the instances of a concern. Such variations are quite common. For example, Bruntink et al. [6] analyze the implementation of a tracing concern in a large company. Despite explicit coding guidelines, the implementation of the tracing concern contained significant variations because of developer mistakes, local optimizations and ambiguous coding guidelines. Concern mining techniques would typically generate separate seeds for each instance variation, delegating the costly task of identifying and merging similar seeds to the developer (step 2 of the mining process).

Examples Figure 1 contains two illustrations of a CCC with small variations in its instances, and two illustrations of a CCC with large variations. In Version 1, the back-end developer accidentally (or intentionally) calls `start_log` instead of `end_log` in the `back_end`. HAM is incapable of merging the concern seeds `start_log` and `end_log` into one seed because the sets of use-sites of both seeds are not identical. The set of use-sites of `start_log` contains `client`, `front_end2` and `back_end`, while the set of use-sites of `end_log` contains `client` and `front_-`

(a) **Version 0.** (b) **Version 1.** (c) **Version 2.** (d) **Version 3.**

Figure 1: Motivating example that illustrates the shortcomings of concern mining techniques. Bold text corresponds to the addition or removal of dependencies on program entities between two successive versions of the example system.

#	CBFA	HAM	Desired
1	start_lock3, start_log, end_log, end_lock3	enqueue, lock, unlock	lock, unlock, enqueue, queue, lock2, unlock2, start_lock3, end_lock3, lock_data_queue, unlock_data_queue
2	unlock, unlock_data_queue, lock_data_queue	lock_data_queue, unlock_data_queue	start_log, end_log
3	lock	lock2, unlock2	N/A
4	enqueue	start_lock3, end_lock3	N/A
5	N/A	start_log	N/A
6	N/A	end_log	N/A

Table 1: Top six results for CBFA (Version 3) and HAM, and the desired mining results for Figure 1.

end2. Breu et al. [5] acknowledge this shortcoming of their technique. CBFA tries to deal with variations in concern instances by taking into account naming conventions, but this often backfires. As the start_lock3, start_log, end_lock3 and end_log functions pair-wise share parts of their name, CBFA assigns all these functions to the first concern seed. However, the *_log and *_lock3 functions are semantically unrelated.

Version 3 of Figure 1 gives two illustrations of a CCC with large variations in its instances. HAM cannot assign start_lock3/end_lock3 and lock2/unlock2 to the same seed, because it does not recognize that the callers of these pairs of functions (i.e., front_end2 and front_end) are actually the same. CBFA's heuristic algorithm is sensitive to variations in the order of functions in the source code. CBFA assigned lock_data_queue to the same seed (#2) as unlock because unlock_data_queue had already been assigned to concern seed #2 (because of the common "unlock") and shares both "data" and "queue" with lock_data_queue. However, the algorithm accidentally processed lock before lock_data_queue, and hence was unable to relate lock to unlock and unlock_data_queue in concern seed #2.

Desired Outcome A concern mining technique should be robust to variations in the instances of a concern. It should be able to identify that start_log and end_log belong to the "logging" concern, and that lock2/unlock2 and start_lock3/end_lock3 belong to the "inter-process communication" concern.

S2: Tendency to ignore important facets of seeds

Motivation Analyzing and completing a seed are crucial steps in the concern mining process, yet concern mining techniques provide developers only with a partial view on possible concerns, i.e., the behavioral facet of a seed (functions), while they ignore the state (variables and types) and any preprocessor entities (macros) of a seed. This makes seeds less representative of actual concerns and slows down the concern mining process, as state and preprocessor entities dominate the source code for procedural languages like C, and state also plays an important role in object-oriented systems (for example in the Singleton and Flyweight design patterns).

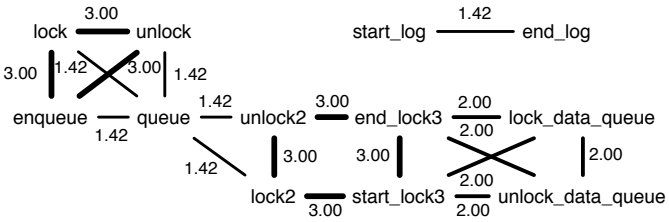

Figure 2: Desired mining results for Figure 1.

Examples In Figure 1, Version 2 introduces synchronization on a `queue` between the `client` and the `front_end`. As this synchronization is implemented differently on the client and the server, different functions are called: the pair `lock`/`unlock` is called in the `client` and the pair `lock2`/`unlock2` is called in the `front_end`. Since the two pairs of functions have different names, neither CBFA, nor HAM are able to link them to each other, even though they operate on arguments of the same type and name. Focusing exclusively on function entities not only makes seeds less rich, it also ignores important hints about the relation between seemingly unrelated program entities.

Desired Outcome If variable and type references were taken into account, an ideal mining technique could find out that the locking function calls in the `client` and the ones in the `front_end` operate on the same global variable as their argument (`queue` of type `queue_t`), which was introduced in the code at the same time. This is a strong indication that `lock`/`unlock` and `lock2`/`unlock2` are part of the same concern (the top desired mining result in Table 1). A richer concern seed is more complete and hence easier to interpret and to expand.

S3: Lack of information about seed relations

Motivation Concern seeds generated by mining techniques are usually just textual lists of code entities, without any relation between the different seeds. It is the developer's task to get a global view on how seeds relate to each other. For example, seeds for a tracing concern and a caching concern are probably independent, whereas the encryption and decryption concerns mentioned in Section 2.1 collaborate to form the "secure communication" composite concern. Because of the large group of developers responsible for the sub-concerns of a composite concern, composite concerns should be documented explicitly.

Examples Neither CBFA, nor HAM are able to detect the relations between the sub-concerns of the "inter-process communication" composite concern in the fourth column of Table 1. CBFA splits the concern across four different seeds, and even mixes it with the logging concern in seed #1. HAM also fragments the concern across four seeds: HAM's top result leads to the `client`-side locking and queue manipulation concern, the second result contains the synchronization concern between `front_end2` and `back_end`, and the third and fourth result refer to two versions of the `front_end2` locking concern.

Desired Outcome An ideal mining technique should identify HAM's top four results, which are scattered across the code, as the sub-concerns of a single composite concern for "inter-process communication". Figure 2 shows the structure of this concern (large graph) and the smaller logging concern (upper right cluster). The different sub-concerns of the former correspond to `client` (upper left cluster), `front_end2` (lower middle cluster) and `back_end` (lower right cluster) functionality.

4. CONCERN MINING USING MUTUAL INFORMATION OVER TIME (COMMIT)

We now present our concern mining technique named COMMIT ("COncern Mining using Mutual Information over Time"), which addresses the aforementioned shortcomings.

General Overview

COMMIT is a history-based concern mining technique. Its architecture is shown in Figure 3. First, COMMIT extracts a report of all change transactions from the subject system's source code repository. A change transaction contains the line numbers of all code lines that have been added, removed or modified by a developer at a particular time. As this information is too low-level, an evolutionary code extractor (C-REX) is used to map added, removed and modified *lines of code* to added and removed calls or references to *program entities* like functions, variables, types and macros [18]. C-REX uses a lexical technique and a number of heuristics to determine the starting and ending line numbers of each function/type-/macro definition and variable declaration. The output of C-REX is a report of changes indicating when a dependency on a particular entity is added or removed.

In the second step, COMMIT determines which function calls, global variable references, macro calls and type references have been added or removed simultaneously in each added, modified, or removed function. Similar to HAM [5], COMMIT suspects that such calls and references belong to the same concern. Co-removal is important because function renaming corresponds to removing calls to the old function, followed by adding calls to the new one. Transactions that change more than 100 entities at once are ignored, as they typically correspond to massive changes of licenses, comments or code formatting [19]. Based on the co-addition and co-removal information, several "seed graphs" are generated with edges between every pair of program entities to which calls or references have been co-added or co-removed over time. As each entity is represented by one node, it belongs to at most one seed graph.

The resulting seed graphs are filtered such that they only contain edges representing *intentional* co-addition and co-removal. To measure how intentional co-additions and co-removals are, COMMIT applies the information-theoretical notion of mutual information [16]. This is a statistical measure of how closely related two entities are, i.e. how often the addition or removal of a dependency (call or reference) on one entity coincides with the addition or removal of a dependency on the other. It is calculated as follows [16]:

$$I(x;y) \quad = \quad log_2\left(\frac{p(x,y)}{p(x)\times p(y)}\right)$$

with

$$p(x) = \frac{\#changed\ entities\ that\ add/remove\ a\ dependency\ on\ x}{total\ \#\ of\ changed\ entities\ in\ history}$$

$$p(y) = \frac{\#changed\ entities\ that\ add/remove\ a\ dependency\ on\ y}{total\ \#\ of\ changed\ entities\ in\ history}$$

$$p(x,y) = \frac{\#changed\ entities\ that\ co\text{-}add/remove\ deps.\ on\ x\ and\ y}{total\ \#\ of\ changed\ entities\ in\ history}$$

A "changed entity" in this definition is an entity (like a function or file) that has been changed in a particular change transaction. Figure 1b contains three changed entities (`client`, `front_end` and `back_end`), whereas Figure 1c contains two (`client` and `front_end`). Applied to concern mining, higher $I(x;y)$ implies

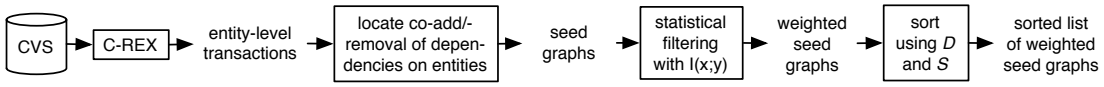

Figure 3: Architecture of COMMIT.

a higher probability that the co-addition and co-removal of dependencies on x and y is intentional. Hence, the higher the probability that entities x and y are part of the same concern.

Once the mutual information $I(x;y)$ between any two co-added and co-removed entities x and y is calculated, we weight each edge in the seed graph with it. Then, we filter out the edge between x and y if its $I(x;y)$ is below a threshold, i.e. if x and y are not related closely enough. This filtering splits up large seed graphs into multiple, disconnected graphs. Figure 2 shows the weighted seed graphs for the desired mining results of Figure 1.

The resulting seed graphs consist of program entities of which dependencies have evolved in close harmony with each other, scattered throughout the system. Hence, we claim that these seed graphs represent concern seeds. Similar to HAM, we order the seeds first on their dimension D, then on their scattering value S. In practice, this means that concern seeds representing CCCs are ranked much higher than seeds representing modular concerns.

Determining the right value for COMMIT's mutual information threshold is an optimization problem [17]. We used a simple, exhaustive search technique to find a solution for the threshold that:

- minimizes the Coefficient of Variation [3] for the dimensions D of the seeds (to avoid extreme situations, such as one giant concern seed or hundreds of small ones);
- maximizes the scattering value S of the reported seeds.

Intuitively, these constraints ensure that the dimension of concern seeds is manageable, while the seeds are as crosscutting, and hence interesting, as possible (high S). We now discuss how COMMIT deals with the three shortcomings outlined in Section 3.

S1: Inability to merge seeds with variations

To deal with variation in concern instances, COMMIT clusters code entities into seeds based on mutual information. COMMIT is robust to small or even large variations in instances as long as the instances co-change sufficiently enough to be statistically valid. In Figure 1, the call to `end_log` is accompanied twice by a call to `start_log`, which links both entities to each other. This link is not invalidated by the fact that `back_end` contains two calls to `start_log`, as the mutual information between `start_log` and `end_log` is still sufficiently high. Similarly, the added calls in `front_end2` are linked to the locking logic of `front_end` in Version 2 because the removal of calls to `lock2`/`unlock2` coincides with the addition of calls to `start_lock3`/`end_lock3`.

S2: Tendency to ignore important facets of seeds

COMMIT considers the co-addition and co-removal of dependencies on all types of program entities (functions, macros, types and variables), not just function calls. In Figure 1, COMMIT is able to link together the locking function calls in `client` and `front_end` (Version 2) based on the common reference to the `queue` global variable. Hence, COMMIT identifies richer concerns than CBFA and HAM, containing behavior, state and preprocessor entities.

S3: Lack of information about seed relations

COMMIT reports a composite concern as a seed graph with multiple interconnected clusters of code entities. As the edges between entities and clusters are weighted by the calculated mutual information values, it is clear which clusters (and hence sub-concerns)

	CBFA			HAM			COMMIT		
	D	S	M	D	S	M	D	S	M
PSQL	16	901	8	33	2	7	35	67	6
NBSD	209	5117	148	49	4	6	147	151	37

Table 2: Average dimension (D), scattering (S) and number of seed entities that are functions (M) for the top twenty concern seeds for PostgreSQL (PSQL) and NetBSD (NBSD).

	CBFA			HAM			COMMIT		
	P	U	C	P	U	C	P	U	C
PSQL	55	40	25	90	50	31	75	75	47
NBSD	90	45	29	75	45	29	75	70	45

Table 3: Precision (P), uniqueness (U) and coverage (C) (in percentage) of the top twenty mining results for CBFA, HAM and COMMIT for PostgreSQL (PSQL) and NetBSD (NBSD).

are strongly related and which ones are not. The reason why COMMIT can reconstruct entire composite concerns is that, during the engineering and maintenance of these concerns, developers make changes to multiple sub-concerns at a time. COMMIT is then able to connect the pair-wise relations between these sub-concerns.

In Figure 1, the changes from Version 1 to Version 2, and from Version 2 to Version 3 gradually introduce the "inter-process communication" concern between `client` and `front_end`, and between `front_end2` and `back_end` respectively. COMMIT is able to reconstruct this composite concern. Its two sub-concerns (client/server and front-end/back-end synchronization) show up as tightly connected sub-graphs on Figure 2. A simple spring-based layout can be used to outline the clusters in a seed graph.

To summarize, we have presented the architecture of COMMIT, and have shown how COMMIT addresses the three identified shortcomings of Section 3. The next section compares COMMIT in a case study to two state-of-the-art concern mining techniques.

5. COMPARATIVE CASE STUDY

We performed a case study to compare the ability of COMMIT, CBFA [39] and HAM [5] to deal with shortcomings S1, S2 and S3. We chose HAM because it is the most closely related history-based technique, and CBFA because it is a recently proposed static mining technique that has been shown to outperform other concern mining techniques. We did not include a dynamic technique, as it is very hard to select representative execution scenarios, especially in the large systems that COMMIT aims at. In addition, CBFA has been shown to perform better than the dynamic Dynamo technique [39]. Section 5.1 presents the setup of the case study and Section 5.2 presents our research hypotheses. We then present the validation results of each research hypothesis.

5.1 Setup of the Case Study

Our case study applies CBFA, HAM and COMMIT on historical revision control data from the open source PostgreSQL [30] database management system (July 1996 to November 2002), and NetBSD [29] operating system kernel and drivers (March 1993 to January 2003). PostgreSQL and NetBSD are large C systems

#	CBFA	HAM	COMMIT
1	error logging	parse/plan tree node types	**retrieve query results and check for errors**
2	α: relation cache management	β: referential integrity constraints management	**preprocess data types in query**
3	α	β	**initialize ODBC configuration**
4	heap tuple management	new and old version of API	**referential integrity triggers in SPI**
5	bool utility functions	β	creating database index
6	data node construction	replay transaction logs	time parsing and decoding
7	invariant checking	γ: SQL transaction support	PL/pgsql execution and debugging
8	memory management	index navigation and manipulation	error handling embedded SQL for C
9	δ: list processing	β	**authenticated database connection**
10	δ	γ	node types of planner

(a)

#	CBFA	HAM	COMMIT
1	ϵ: multi-platform Linux emulation	ζ: ioctl/termios flags translation	**multi-platform ISO/IP IPC**
2	/proc/sys support	duplication in old/new ARM boot loader	ϵ
3	η: device driver API	basic error/log/input/... handling for USB	**SCSI controller driver logic**
4	error handling	ζ	Digital/Intel 21x4x ("Tulip") Ethernet driver
5	ϵ	ζ	**multi-platform (pseudo-)terminal handling**
6	panic handling	ζ	Advanced Systems Inc. SCSI controller flags
7	interrupt priority level	signal flags translation	NFS file system
8	η	VAX device switch table	Gravis UltraSound audio cards driver
9	DDB in-kernel debugger	Amiga CyberVision 64 parameters	ϵ
10	η	ISA/ATA data transfer	Coda file system

(b)

Table 4: The top ten mining results for CBFA, HAM and COMMIT on (a) PostgreSQL and (b) NetBSD. Dark cells correspond to CCCs, and bold text to composite concerns. Greek symbols indicate duplicate concerns.

(over 800 kLOC and 2 MLOC respectively) with a long, well-documented development history (6,199 and 36,635 change transactions respectively). Their size ensures that our work scales to large, long-lived systems. Their accessible history permits us to easily verify many of our identified concerns. Their distinct domains ensure that our technique applies to different software systems. Since CBFA operates on a single version of a software system, we apply it on the PostgreSQL version of November 2002 and the NetBSD version of January 2003.

We re-implemented HAM and CBFA because the HAM prototype did not support C code and the CBFA prototype did not scale to large systems [39]. To make our comparison fair, we extended CBFA and HAM to also take into account variables, types and macros, in addition to functions. We also automatically filter out those concern seeds containing only standard macro constants (e.g., `NULL`), types (e.g., `u_int`) and utility functions (e.g., `printf`). As the extended re-implementations consistently performed better than the non-extended ones, we only report the results of the former. Using the search-based heuristics for COMMIT's threshold (Section 4), we obtained a mutual information threshold of 10.5 for PostgreSQL and 14.5 for NetBSD. The differences between these thresholds are caused by the larger number of changes we considered for NetBSD (36,635 compared to 6,199 for PostgreSQL). We empirically determined that a minimum similarity threshold of 0.4 between the names of two entities gives better results for CBFA than the threshold used in [39].

Developers do not have the patience to sort through all generated seeds to find a relevant CCC. For this reason, our case study considers only the top twenty mining results of the three mining techniques. The first two authors independently analyzed the six lists of top twenty seeds (3 techniques for 2 subject systems) to determine whether the seeds lead to a CCC, and, possibly, a composite concern. Modular concerns were interpreted as undesired mining results, as one does not need concern mining techniques to locate them. Afterwards, both authors reconciled their findings to come up with a unified list of CCCs per technique and subject system. Conflicts were resolved using information from the source code, online documentation, mailing lists and change logs. This process is similar as to how mining techniques are used in practice, as the interpretation of mining results is a subjective process [25].

The identified concerns are summarized in Table 4, and the corresponding metrics in Table 2. Due to space limitations, Table 4 reports only the top ten results of PostgreSQL and NetBSD. Seeds identified as CCCs have a gray background. Seeds that lead to the same concern ("duplicate" seeds) are marked with Greek symbols: the first occurrence of a duplicate seed contains a Greek symbol and the name of the concern, whereas all repetitions of the seed only contain the Greek symbol. For ease of comparison, Table 2 reports the scattering S for all mining techniques. We note that the CBFA seeds in Table 4 are sorted using the cluster Fan-in metric, as recommended in the original paper [39], and not using S.

5.2 Research Hypotheses

The goal of the case study is to validate that COMMIT addresses the three shortcomings of Section 3. We formulate this goal in the form of three research hypotheses:

H1 COMMIT identifies a larger number of unique CCCs.

H1 evaluates the ability of COMMIT to deal with small variations in concern instances (shortcoming S1).

H2 COMMIT returns richer seeds.

H2 claims that state and preprocessor entities represent a significant part of seeds (shortcoming S2).

H3 COMMIT complements CBFA and HAM.

> H3 states that COMMIT's ability to identify relations between seeds, in particular between the sub-concerns of a composite concern (shortcoming S3), makes COMMIT complementary to CBFA and HAM.

For each hypothesis, we discuss its motivation, its validation approach, and our findings on the PostgreSQL and NetBSD data.

H1: COMMIT identifies a larger number of unique CCCs

Motivation Concern mining techniques return a large number of concern seeds. Several seeds may refer to the same concern, as mining techniques are unable to resolve small variations in instances of a concern (S1). This leads developers to waste their time and effort merging or eliminating duplicate seeds. An approach that returns a large number of unique CCCs is preferred over an approach that might return a larger set of seeds that eventually contain duplicate seeds.

Approach For each mining technique, we measure the **uniqueness** U of seeds using the following formula:

$$Uniqueness\, U = \frac{\#\ of\ unique\ CCCs\ identified}{20}$$

The obtained percentages are shown in Table 3. The uniqueness metric differs from the classical **precision metric** P in the following way: P divides by the number of *unique* seeds instead of the *total* number of seeds (20). Intuitively, if a technique would return 20 duplicate seeds of one unique CCC, this would yield a precision of 100% since all returned seeds are valid CCCs. However, the actual value for human concern miners is really just 5%, because they wasted a great deal of time interpreting 19 duplicate seeds. Table 3 reports P to give an idea about the raw false positive rate of the three concern mining techniques.

Findings Table 3 shows that COMMIT finds more unique CCCs than CBFA and HAM. The results for HAM are similar to previous experiments on Java systems [5].

The higher percentage of unique CCCs identified by COMMIT relative to CBFA and HAM can be explained by the clustering technique used by the three techniques: CBFA clusters based on the name of entities, HAM based on the set of callers of a function and COMMIT based on the mutual information between entities. With CBFA, entities with totally different names always end up in different seeds, whether or not they are related. HAM has a relatively low U due to the high overlap of entities in the top twenty seeds (seed #9 in PostgreSQL is even a subset of seed #3). Entities can belong to multiple seeds if dependencies on them have been added or removed in different time periods or by different developers [5].

Although COMMIT finds three more false positives than HAM in PostgreSQL (difference of 15% in precision P), its U is considerably higher than for HAM and CBFA. COMMIT merges seeds as soon as dependencies on at least one entity in each seed have been co-added or co-removed a statistically significant number of times. In addition, each entity can only belong to at most one seed, otherwise COMMIT would merge two seeds into one.

COMMIT identifies a larger number of unique CCCs, and hence addresses shortcoming S1.

H2: COMMIT returns richer seeds

Motivation In the third step of the concern mining process, developers need to rebuild the full concern from the basic seed returned by a concern mining technique. Richer seeds containing state and preprocessor entities in addition to behavior, provide developers with better insights into the semantics of a seed.

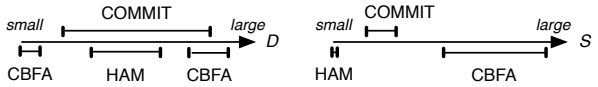

Figure 4: The dimension D and scattering S of the CBFA, HAM and COMMIT results complement each other.

Approach To measure the richness of a concern seed, we measure the dimension D of the top twenty concern seeds, and the **number of functions** M in each of these seeds. We use the latter to demonstrate that functions represent only a minor part of CCCs.

Findings Table 2 shows, for the three mining techniques, the average seed dimension D and number M of seed entities that are functions. On average, COMMIT and HAM yield seeds with more than 30 entities for PostgreSQL, and COMMIT and especially CBFA yield seeds of more than 100 entities for NetBSD. On average, 10% to 25% of the seed entities generated by HAM or COMMIT are functions, whereas for CBFA this is 50% to 70%.

The extremely low percentage of functions in the concern seeds for HAM and COMMIT clearly shows that a significant facet of CCCs is formed by variables, types, macros and conditional compilation, especially in C systems. Some of the CCCs do not even involve functions, such as seeds #1 and #4 of HAM (PostgreSQL) and seed #10 of COMMIT (PostgreSQL). Across the three techniques, five CCCs of PostgreSQL and eleven CCCs of NetBSD would go unnoticed if the concern mining techniques would not take into account the state of concerns and preprocessor entities.

The large dimension of CBFA seeds for NetBSD follows from the naming convention and processing order limitations discussed in Section 3. The high percentage of functions is due to the Linux emulation API and the device driver API, both of which consist of many functions with a strictly enforced naming scheme.

The impact of shortcoming S2 is significant, as functions represent only 10% to 25% of the entities in the seeds generated by HAM and COMMIT. Without considering state and preprocessor entities, 16 CCCs would not even be detected.

H3: COMMIT complements CBFA and HAM

Motivation Concern mining is time and effort consuming. As each technique has its own strengths, combining multiple techniques yields a more complete and diverse set of CCCs [9]. We hypothesize that COMMIT's ability to identify relations between seeds (S3), in particular between the sub-concerns of composite concerns, complements the strengths of existing mining techniques.

Approach As the complete set of CCCs in a system is very hard to determine, we measure the **concern coverage** C [38] (Table 3):

$$Coverage\, C = \frac{\#\ of\ unique\ CCCs\ identified}{total\ \#\ of\ unique\ CCCs\ across\ all\ techniques}$$

This gives an indication about the completeness and overlap of the results of a given technique compared to all obtained mining results. In addition, we use the dimension D and scattering S (Table 2) of the seeds to characterize them, and we qualitatively examine the seeds (Table 4).

Findings The mining results of CBFA, HAM and COMMIT are complementary. The concern coverage of COMMIT is 55.2% to 88% higher than the coverage of CBFA, and 51.6% to 55.2% higher

	CBFA	HAM	COMMIT
PostgreSQL	0.76	1.0	0.76
NetBSD	1.0	0.66	1.0

Table 5: Cohen Kappa inter-rater agreement for the two first authors for the 6 lists of top twenty results.

than the coverage of HAM (Table 3). There is only one CCC shared between techniques for PostgreSQL (seeds #18 and #19 of HAM are similar to seed #3 of COMMIT) and one for NetBSD (seeds #1 and #5 of CBFA are similar to seeds #2 and #9 of COMMIT). In addition to concern coverage, Figure 4 shows that COMMIT fills a missing gap in concern mining techniques, because CBFA specializes in small or very large, homogeneous CCCs (low or very high D) with high scattering (high S), HAM specializes in large, homogeneous CCCs with low scattering, and COMMIT specializes in large, heterogeneous CCCs with medium scattering. Hence, it makes sense to combine the three mining techniques.

As an example, we found that many of the seeds returned by COMMIT for NetBSD correspond to individual drivers. Most of the drivers correspond to modular concerns, except for drivers that are generic (default) for a large class of devices, such as the Western Digital SCSI drivers in seed #3. These drivers turned out to be non-modular composite concerns, in the sense that many of their instances were scattered across the implementation of the kernel. CBFA, on the other hand, is able to find seeds for the general device driver API, across all drivers, because the API functions have similar names. COMMIT does not find a high mutual information value for the API functions of different drivers, as the majority of drivers have a modular implementation and are developed independently. CBFA and COMMIT clearly complement each other.

Our qualitative analysis showed that the CCCs identified by CBFA tend to be a mix of development-oriented (e.g., error logging, tracing, invariant checking and memory management) and domain-specific concerns (e.g., relational table management in PostgreSQL and device driver API in NetBSD), whereas the ones identified by HAM and COMMIT are all closer to the domain of database systems or operating systems, like referential integrity constraints, transaction management and data layout in PostgreSQL, and terminal management and data transfer in NetBSD.

Only COMMIT is able to recover composite concerns. Six of the seeds identified by COMMIT for PostgreSQL (especially the top four seeds) lead to composite concerns, with the number of sub-concerns varying between two and six. Similarly, five of the seeds for NetBSD lead to composite concerns, with the number of sub-concerns also varying between two and six. CBFA and HAM only find single sub-concerns of these composite concerns, such as CBFA seeds #1 and #5 for NetBSD and HAM seed #19 for PostgreSQL. CBFA and HAM are not able to reveal the relations between these sub-concern seeds. This makes it difficult for developers to identify and reconstruct the full composite concern.

We now discuss PostgreSQL seed #1 for COMMIT in more detail, as this is one of the best illustrations of a composite concern that we found. Concern #1 is related to the ODBC framework ("Open DataBase Connectivity"), which is a cross-platform and cross-language API specification to shield client programs from differences in relational database technology. Client programmers write their systems in terms of ODBC, after which ODBC transforms the generic queries into database-specific ones using a database-specific ODBC driver. Concern #1 is responsible for "retrieving query results and checking for errors". It consists of the following six co-operating sub-concerns:

1. Connection configuration concern, which configures the ODBC connection from a client program.
2. Data retrieval concern, which supports the retrieval of data from the database over the established connection.
3. SQL-to-ODBC mapping concern, which permits the efficient mapping of SQL query results to ODBC-specific types.
4. Type conversion concern, which is used to map ODBC-specific types to general C types.
5. Error handling concern for type conversions.
6. Error handling concern for client connectivity.

CBFA, HAM and COMMIT complement each other. CBFA identifies small or very large, development-oriented concerns with high scattering. HAM identifies large, domain-specific concerns with low scattering. COMMIT identifies large, domain-specific concerns with medium scattering, many of which are composite concerns (shortcoming S3).

6. THREATS TO VALIDITY

Construct Validity checks whether we use the right metrics in our study. This paper focuses on macro-level shortcomings of concern mining techniques, i.e. variation of seeds, ignored facets of seeds and relations between seeds. However, mining techniques also have micro-level shortcomings [26], such as false positive and negative entities inside seeds.

Internal Validity is concerned with finding out whether other plausible hypotheses can explain our findings. Our comparative case studies rely on re-implemented and extended versions of HAM and CBFA. HAM had not been applied to non-object oriented systems before, but our findings confirmed earlier object oriented studies [5]. Our extensions to HAM and CBFA performed significantly better than the original algorithms on our subject systems, hence we only reported the performance of the former re-implementations.

To analyze the source code changes, C-REX uses a lexical technique that does not resolve pointers or multiple definitions of the same function [18]: if two functions have the same name, C-REX does not know which one is called. However, as entities with the same name in general represent the same concern (cf. polymorphism in object-oriented systems [25]), COMMIT treats all called or referenced entities with the same name as one.

The optimization criteria of Section 4 to determine COMMIT's threshold encode our intended usage of COMMIT, i.e., getting a high-level overview of concerns and their structure. If developers want to zoom in on very specific concerns, other criteria should be used, resulting in a higher threshold. The ability to tweak COMMIT based on the focus of the concern miner, is a powerful feature.

Finally, our assumption to ignore changes of more than 100 entities because such changes typically correspond to massive changes of, for example, licenses, might not always hold.

External Validity deals with the generalization of our results. For our study, we chose two large, long-lived legacy C systems, similar to [1, 2, 6, 7, 15, 34, 39], and compared the performance of COMMIT to the performance of CBFA and HAM. We chose PostgreSQL and NetBSD because of their size and long history, and respectively considered 7 and 11 years of that history, due to the amount of effort to carefully analyze concerns across the lifetime of such large systems. The development process (open versus closed source), programming language and domain of the applications are likely to affect the results and the applicability of COMMIT.

Comparisons with other concern mining techniques are needed to further generalize our findings, in particular on smaller de facto benchmark systems like JHotDraw [9], but in this paper we focused

on comparing COMMIT to two other techniques on larger-scale, long-lived software systems. Because our CBFA re-implementation consists of a Fan-in [25] re-implementation and a clustering tool, we were able to compare Fan-in to COMMIT. As Fan-in did not benefit from our extension to support non-function entities, we used the original algorithm. The seed dimension D is always 1, whereas the scattering S is significantly lower than for CBFA. The uniqueness U and coverage C are slightly worse than for CBFA. Hence, COMMIT also complements the results of Fan-in.

Reliability Validity reflects the degree to which someone doing the same study would reach the same results. It is a known fact that concern mining is inherently subjective [26]. To counter this, the two first authors independently validated the top twenty results for CBFA, HAM and COMMIT (similar to for example [32]), after which their findings were reconciled. The Cohen Kappa inter-rater agreements [3] for the decision if a concern seed is a CCC are shown in Table 5. In three cases, the two raters agreed "perfectly", whereas in the other three cases the agreement was still "substantial": in two cases (agreement of 0.76) the raters disagreed on two seeds, and in one case (agreement of 0.66) the raters disagreed on three seeds. Per subject system, the interpretation of all 60 mining results took ninety hours in total, primarily because none of the authors knew the internals of PostgreSQL or NetBSD before this case study, and the seeds of HAM and COMMIT contain entities from different periods in time. We cannot make any claims about the duration of our analysis for individual techniques because of the learning effect involved with the subject systems.

7. RELATED WORK

None of the existing concern mining or browsing techniques addresses all three shortcomings. We picked CBFA and HAM as example techniques to demonstrate this. We first discuss concern mining techniques, then concern browsing techniques.

Various static concern mining techniques have been proposed: Fan-in value [25], identifier analysis [36, 39], clone detection [7, 34], random walks [38], Latent Dirichlet Allocation [1], clustering [12, 39] or even a mix of techniques [33]. The techniques based on inexact equality like clone detection techniques [7, 34] and PAM [38] deal better with slight variations in concern instances than CBFA and HAM, but are not able to handle larger variations like near clones (S1). Concern mining techniques tend to focus only on functions (S2), except for clone detection techniques [7] and CBFA. To our knowledge, no concern mining technique reports relations between concern seeds (S3).

Dynamic concern mining techniques [4, 15, 20, 35] use execution scenarios to detect which program entities collaborate in a use case. While this theoretically allows them to exercise multiple sub-concerns of a composite concern, one has to make sure that the executed scenarios include all sub-concerns. This problem makes it hard in general to detect all instances of a concern.

History-based concern mining techniques [5, 8] analyze which program entities change together frequently, as such entities likely belong to the same concern. As illustrated for HAM, the use of exact equality checks between sets of calling entities makes these techniques suffer from S1 and S2. Canfora et al. [8] tackle S2 by considering individual lines of code instead of just function calls.

Concern browsing techniques [2, 21, 23, 27, 31, 32, 37] do not generate *all* concern instances up front, but suggest related code fragments as a developer browses through the source code in her IDE. Manual input is needed to deal with shortcomings S1 and S2, and the detection of relations between seeds is still troublesome. Some techniques [27, 31] manage relations between concerns by allowing users to manually structure concerns in a hierarchy.

8. CONCLUSION

Given the high maintenance and re-engineering risks represented by crosscutting concerns, especially composite concerns, automatic identification of these concerns is an important task. This paper presents a history-based concern mining technique named COMMIT ("COncern Mining using Mutual Information over Time") that addresses three major shortcomings in the concern mining process. First, it uses a robust, statistical clustering mechanism to deal with small and even large variations in the instances of a concern. Second, COMMIT considers dependencies on behavior (functions), state (variables and types) and preprocessor entities (macros) to identify different facets of seeds. Third, COMMIT uncovers the relations between concern seeds. In particular, it can detect composite concerns with multiple co-operating sub-concerns.

COMMIT has been applied on the open source PostgreSQL database and NetBSD operating system kernel to compare its performance with two state-of-the-art mining techniques, i.e. CBFA [39] and HAM [5]. We found that for the top twenty results of each technique, COMMIT obtained up to 87.5% more unique concerns and up to 88% higher concern coverage, even after adapting CBFA and HAM to non-function C entities. COMMIT's ability to detect highly heterogeneous, domain-specific CCCs makes it complementary to CBFA, which focuses on small or very large, homogeneous CCCs with high scattering, and to HAM, which focuses on large, homogeneous CCCs with low scattering.

The take-home message of this paper is that wide-spread adoption of concern mining techniques for software maintenance, reverse-engineering and re-engineering is only possible if they:

1. incorporate all possible sources of data (such as change history and execution traces) and information (such as state, behavior and preprocessor entities);
2. deal with noise in this data and information;
3. provide more context about seeds to improve the manual concern mining steps.

Our results encourage us to study hybrid concern mining techniques, supported by statistical analysis.

Acknowledgments We would like to thank the anonymous reviewers and Nicolas Bettenburg for their valuable comments.

9. REFERENCES

[1] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *Proc. of the 23rd ACM SIGPLAN conf. on Object-oriented programming systems languages and applications (OOPSLA)*, pages 543–562, 2008.

[2] E. L. A. Baniassad and G. C. Murphy. Conceptual module querying for software reengineering. In *Proc. of the 20th Intl. Conf. on Software Engineering (ICSE)*, pages 64–73, April 1998.

[3] S. Boslaugh and P. Watters. *Statistics in a Nutshell: A Desktop Quick Reference*. O'Reilly Media, Inc., 1st edition, August 2008. 476 p.

[4] S. Breu and J. Krinke. Aspect mining using event traces. In *Proc. of the 19th Conf. on Automated Software Engineering (ASE)*, pages 310–315, Linz, Austria, September 2004.

[5] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proc. of the 21st IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*, pages 221–230, 2006.

[6] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In

Proc. of the 6th Intl. Conf. on Aspect-Oriented Software Development (AOSD), pages 199–211, March 2007.

[7] M. Bruntink, A. van Deursen, T. Tourwé, and R. van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proc. of the 20th Intl. Conf. on Software Maintenance (ICSM)*, pages 200–209, September 2004.

[8] G. Canfora, L. Cerulo, and M. Di Penta. On the use of line co-change for identifying crosscutting concern code. In *Proc. of the 22nd IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 213–222, 2006.

[9] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Control*, 14(3):209–231, 2006.

[10] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, 26(5):88–98, 2001.

[11] T. Cohen and J. Y. Gil. AspectJ2EE = AOP + J2EE: Towards an aspect based, programmable and extensible middleware framework. In *Proc. of the 18th European Conf. on Object-Oriented Programming (ECOOP)*, volume 3086, pages 14–18, 2004.

[12] G. S. Cojocar and G. Czibula. On clustering based aspect mining. *Proc. of the 4th Intl. Conf. on Intelligent Computer Communication and Processing (ICCP)*, pages 129–136, Aug. 2008.

[13] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proc. of the 3rd intl. conf. on Aspect-oriented software development (AOSD)*, pages 56–65, 2004.

[14] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515, 2008.

[15] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Software Eng.*, 29(3):210–224, 2003.

[16] R. M. Fano. *Transmission of Information: A Statistical Theory of Communication*. MIT Press, 1961.

[17] M. Harman. The current state and future of search based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 342–357, 2007.

[18] A. E. Hassan. *Mining Software Repositories to Assist Developers and Support Managers*. PhD thesis, University of Waterloo, Waterloo, ON, Canada, 2004.

[19] A. E. Hassan. Automated classification of change messages in open source projects. In *Proc. of the 2008 ACM Symp. on Applied computing (SAC)*, pages 837–841, 2008.

[20] L. He and H. Bai. Aspect mining using clustering and association rule method. *Intl. Journal of Computer Science and Network Security*, 6(2):247–251, February 2006.

[21] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proc. of the 22nd IEEE/ACM intl. conf. on Automated Software Engineering (ASE)*, pages 14–23, 2007.

[22] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, IV(LNCS 4640):143–162, 2007.

[23] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proc. of the 4th intl. conf. on Aspect-oriented software development (AOSD)*, pages 159–168, 2005.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of the 11th European Conf. on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242, 1997.

[25] M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Trans. Softw. Eng. Methodol.*, 17(1), January 2008.

[26] K. Mens, A. Kellens, and J. Krinke. Pitfalls in aspect mining. In *WCRE '08: Proc. of the 2008 15th Working Conf. on Reverse Engineering*, pages 113–122, 2008.

[27] K. Mens, B. Poll, and S. González. Using intentional source-code views to aid software maintenance. In *Proc. of the 19th Intl. Conf. on Software Maintenance (ICSM)*, pages 169–178, 2003.

[28] K. Mens and T. Tourwé. Evolution issues in aspect-oriented programming. In T. Mens and S. Demeyer, editors, *Software evolution*, chapter 9, pages 203–232. Springer Verlag, 1st edition, February 2008.

[29] NetBSD. http://www.netbsd.org/.

[30] PostgreSQL. http://www.postgresql.org/.

[31] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, pages 406–416, May 2002.

[32] D. Shepherd, Z. P. Fry, E. Hill, L. L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proc. of the 6th Intl. Conf. on Aspect-Oriented Software Development (AOSD)*, pages 212–224, March 2007.

[33] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll. Timna: a framework for automatically combining aspect mining analyses. In *Proc. of the 20th IEEE/ACM intl. Conf. on Automated software engineering (ASE)*, pages 184–193, 2005.

[34] N. Singh, C. Gibbs, and Y. Coady. C-CLR: A tool for navigating highly configurable system software. In *Proc. of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), AOSD*, 2007.

[35] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proc. of the 11th Working Conf. on Reverse Engineering (WCRE)*, pages 112–121, November 2004.

[36] T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In *Proc. of the 4th IEEE Intl. Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 97–106, September 2004.

[37] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proc. of the 25th Intl. Conf. on Software Engineering (ICSE)*, pages 408–418, 2003.

[38] C. Zhang and H.-A. Jacobsen. Efficiently mining crosscutting concerns through random walks. In *Proc. of the 6th Intl. Conf. on Aspect-Oriented Software Development (AOSD)*, pages 226–238, March 2007.

[39] D. Zhang, Y. Guo, and X. Chen. Automated aspect recommendation through clustering-based fan-in analysis. In *Proc. of the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*, pages 278–287, Sept. 2008.