Concept Location using Formal Concept Analysis and Information Retrieval

DENYS POSHYVANYK and MALCOM GETHERS
The College of William and Mary
and
ANDRIAN MARCUS
Wayne State University

The paper addresses the problem of concept location in source code by proposing an approach that combines Formal Concept Analysis and Information Retrieval. In the proposed approach, Latent Semantic Indexing, an advanced Information Retrieval approach, is used to map textual descriptions of software features or bug reports to relevant parts of the source code, presented as a ranked list of source code elements. Given the ranked list, the approach selects the most relevant attributes from the best ranked documents, clusters the results, and presents them as a concept lattice, generated using Formal Concept Analysis.

The approach is evaluated through a large case study on concept location in the source code on six open-source systems, using several hundred features and bugs. The empirical study focuses on the analysis of various configurations of the generated concept lattices and the results indicate that our approach is effective in organizing different concepts and their relationships present in the subset of the search results. In consequence, the proposed concept location method has been shown to outperform a standalone Information Retrieval based concept location technique by reducing the number of irrelevant search results across all the systems and lattice configurations evaluated, potentially reducing the programmers' effort during software maintenance tasks involving concept location.

Categories and Subject Descriptors: **D.2.7 [**Software Engineering**]:** Distribution, Maintenance, and Enhancement – enhancement, restructuring, reverse engineering, and reengineering
General Terms: Documentation, Design
Additional Key Words and Phrases: Concept location, feature identification, Information Retrieval, Formal Concept Analysis, program comprehension, software evolution and maintenance

ACM File Format:
POSHYVANYK, D., GETHERS, M., MARCUS, A., 2010. Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology (TOSEM).*

1. INTRODUCTION

Identifying the parts of the source code that correspond to specific system functionality is a prerequisite to program comprehension and is one of the most common activities undertaken by developers. This process is called *concept* (or *feature*) location [Wilde et al. 1992] and it is among the key questions that developers ask during software development and maintenance [Sillito et al. 2008]. During software evolution, the existing code is modified in order to add new features or to alter existing ones. This is the context in which we address concept location. During the incremental change of software [Rajlich and Gosavi 2004], developers use concept location to identify the location in the source code where they will implement the changes. The input is usually a change request (or a bug report) and the result is the location of the change (for example, a method in source code). Although the goal of incremental change is to identify all the components that need to be changed, the developer must find the location in the code where the first change will be made. For that, programmers search the whole program and various concept location techniques aim at narrowing down this search space. Recent research literature defines this step as finding only one part of the concept

implementation, which is the starting point of a change [Gay et al. 2009; Liu et al. 2007; Poshyvanyk et al. 2007; Revelle et al. 2010]. The full extent of the change is then handled by impact analysis, which is methodologically different from concept location, and thus is treated separately in the research literature [Rajlich and Gosavi 2004]. In this paper, we explicitly address the identification of methods in software that are part of the implementation of a feature (that is, they are changed when the feature is altered) and can be used as starting points for impact analysis. *Concept* location is also referred to in the literature as *feature* identification or *concern* location. Section 3 discusses in more detail the distinctions between features and concepts. Through the rest of the paper we use the term *concept location*, even when we refer to similar techniques that are named differently. When the context may generate confusion between the uses of the word *concept* in *concept location* vs. *concept analysis* we use *feature* instead of *concept*.

One of the most commonly used techniques to support concept location is based on text search in source code, where developers write queries and a search engine returns a list of source code elements relevant to the query. In many cases, only a small fraction of the search result is actually relevant to the concept being located. In these cases, developers either undertake the daunting task to investigate in detail as much as they can from the results, or they reformulate their query to reduce the size of the search results. Eventually, even after a series of refined queries, the user will need to investigate a subset of the results. This type of concept location is an instance of a text retrieval task. In general, it is common in text retrieval and data mining applications to either cluster the data prior to executing queries or to cluster the results after a query is executed [Carpineto et al. 2009]. Parts of the software can relate to each other in more than one way (*e.g.*, via data flow, control flow, conceptual similarity, past co-changes, etc.). In consequence it is hard to find a similarity measure that supports partitioning of the entire software in clusters relevant to all of its features. Another problem of pre-clustering the search space is in establishing the appropriate number of clusters. Given these issues, our choice is to cluster part of the search results, which are already relevant to the concept at hand. Our work aims at helping developers in reducing their efforts by providing additional structure in the search results, such that parts of the source code are grouped based on common topics. Similar approaches are also employed by popular internet search engines, such as, Vivisimo[1] and Yippy[2].

Specifically, we augment an existing information retrieval (IR) based technique for concept location [Marcus et al. 2004] with automatic clustering of the search results using formal concept analysis (FCA). This approach is inspired by previous work, which applied FCA to free-text search [Cigarrán et al. 2004]. The IR-based concept location technique uses a search engine based on Latent Semantic Indexing (LSI) [Deerwester et al. 1990], which allows users to search source code and related textual documentation by writing natural language queries and retrieving a list of source code elements (that is, classes, methods, functions or files), which are ranked based on their similarity to the query. Based on the ranked results of the search, we automatically generate a labeled concept lattice. Developers can determine whether a node from the concept lattice (that is, a topic or category) is relevant or not to their query by simply examining its label; they can then explore only relevant nodes in the lattice and ignore the other ones, thus reducing their search effort.

[1] http://vivisimo.com/ (*verified on 08/18/2011*)
[2] http://search.yippy.com/ (*verified on 08/18/2011*)

2. BACKGROUND

In this section we present background information on FCA, a mathematical technique for analyzing binary relations and LSI, an advanced information retrieval method. These two techniques are at the core of our work and come from research fields outside software engineering.

2.1 Formal concept analysis

Formal concept[3] analysis [Ganter and Wille 1996] is a branch of mathematical lattice theory that provides means to identify meaningful groupings of *objects*[4] that share common *attributes* and it also provides a theoretical model to analyze hierarchies of these groupings.

The main goal of FCA is to define a *concept* as a unit of two parts: *extension* and *intension*. The extension of a concept covers all the objects that belong to the concept, while the intension comprises all the attributes, which are shared by all the objects under consideration.

In order to apply FCA, the formal context or incidence table of objects and their respective attributes is necessary. The formal context consists of a set of objects O, a set of attributes A, and a binary relation $R \subseteq O \times A$ between objects and attributes, indicating which attributes are possessed by each object. Formally, it can be defined as $C = (A, O, R)$. From the formal context, FCA generates a set of concepts where every concept is a maximal collection of objects that possess common attributes. More formally, a concept is a pair of sets (X, Y) such that:

$$X = \{o \in O \mid \forall a \in Y: (o,a) \in R\}$$
$$Y = \{a \in A \mid \forall o \in X: (o,a) \in R\}, \text{ where}$$

X is considered to be the *extension* of the concept and Y is the *intension* of the concept. This set of concepts form a *complete partial order* where some concepts are super- or sub-concepts with respect to others.

The set of all concepts constitutes a concept lattice and there are several algorithms to compute concepts and concept lattices from a given formal context. For details on these algorithms as well as more complete description on FCA, we refer to the work of Ganter and Wille [Ganter and Wille 1996]. It should be noted that FCA provides intensional descriptions for concept nodes, thus improving comprehensibility of concept lattices. Additionally, an inherent advantage of concept lattices is a set of relations among the concept nodes, which users can navigate in a lattice. Among visible disadvantages of FCA, the size and complexity of lattices depends on the number of objects and attributes and can be computationally demanding as compared to other clustering techniques [Cigarrán et al. 2004].

2.2 Latent Semantic Indexing

In the proposed concept location approach we utilize an information retrieval method, LSI [Deerwester et al. 1990], as a text indexing and search engine. LSI is based on the Vector Space Model (VSM) [Salton and McGill 1983] IR technique, commonly used in text retrieval. In such application, the retrieved data is (natural) text, which is organized

[3] Note the difference between the use of the term *concept* in FCA and concept location
[4] Note the difference between the terms *object* and *attribute* in FCA and OOP

into documents. The words that occur in a document are considered as its defining features. For example, a word (referred to as a term, when it is not part of a language) that occurs often in a document *d* and rarely in others, is specific to *d* (that is, it is essential to capture the semantics of *d* and not of other documents). VSM (and implicitly LSI) represent documents as vectors in a space that spans the words encountered in a corpus. This allows to define and to compute textual similarities between documents, based on their mathematical representation as vectors. For example, the cosine between two vectors is often used as a similarity measure for their corresponding documents.

LSI was originally developed in the context of information retrieval as a way of overcoming problems with polysemy and synonymy that occurred with VSM [Salton and McGill 1983] approaches. Some words appear in the same contexts (synonyms) and an important part of word usage patterns is blurred by accidental and inessential information. The method used by LSI to capture the vital semantic information is dimension reduction, selecting the most important dimensions from a co-occurrence matrix decomposed using singular value decomposition (SVD). As a result, LSI offers a way of assessing semantic similarity between any two samples of text in an automatic, unsupervised way. LSI does not utilize a grammar or a predefined vocabulary; however, it can use a list of stop-words that can be extended by the user. Stop-word lists identify words which should be excluded from the corpus. LSI is based on a Singular Value Decomposition (SVD) of the term-by-document vector space derived from the corresponding co-occurrence matrix. SVD is a form of factor analysis (a statistical method capable of reducing dimensionality of a dataset), which is used to reduce dimensionality of the feature space to capture most essential semantic information. The formalism of SVD is rather lengthy to be presented in the paper, thus we refer the reader for complete details elsewhere [Deerwester et al. 1990]. The original vectors are then projected on a smaller space, determined using SVD. This projection allows representing both words and documents in the same space; hence LSI allows computing similarities between terms and documents. This is important in our application as we use these similarities to find the terms in each document, which will be used as *attributes* by Formal Concept Analysis.

Originally LSI has been applied on natural language corpora, however, the method has been shown to lend itself well to other types of data, for example, textual information extracted from source code and associated documentation. Some of the software engineering problems, related to concept location, which have been addressed using LSI are: traceability link recovery between source code and documentation [De Lucia et al. 2007; Jiang et al. 2008; Marcus et al. 2005a], tracing requirements [Hayes et al. 2006; Lo et al. 2006] and other software artifacts [Lormans and Van Deursen 2006], identifying clones in software [Marcus and Maletic 2001; Tairas and Gray 2009], retrieving relevant artifacts in project histories [Cubranic et al. 2005], measuring coupling [Poshyvanyk et al. 2009] and cohesion [De Lucia et al. 2008; Marcus et al. 2008] of classes. In these applications, the documents are formed using the source code (that is, a document can be a class, method, function, package, etc.) or external documentation. The terms are the words and identifiers that occur in the documents.

The various LSI-based concept and feature location techniques follow the following steps [Gay et al. 2009; Liu et al. 2007; Marcus et al. 2004; Poshyvanyk et al. 2007; Poshyvanyk and Marcus 2007]. These steps include: extraction of textual information from the source code; pre-processing and partitioning of text into a corpus of documents; indexing the corpus; formulating and executing queries; and analyzing and inspecting ranked lists of the results. We describe some of these steps in more detail in Section 4, in the context of our proposed approach.

3. RELATED WORK

This section outlines research related to our work, where we present existing approaches to feature and concept location, with specific focus on the use of FCA in this context.

Concept location is also referred to in the literature as *feature* identification or *concern* location. *Features* are special concepts (that is, a subset of concepts) that are associated with the user visible functionality of the system. The shared goal of these techniques is to identify the source code units (such as, methods, function, classes, etc.) that implement (part of) a concept of interest from the problem or solution domain of the software. Concept location is an essential part of the software change process [Rajlich and Gosavi 2004].

Existing approaches to concept location use different types of software and data analyses. They can be broadly classified into static, dynamic, and combined analysis based approaches [Dit et al. 2011b]. Our approach is a static technique, as it uses only the source code (and possibly documentation) without executing the software. We will focus this section on static concept location techniques alone. Based on the information they use for analysis, the static techniques can be further refined into: text-based techniques, which use text searching; structural-based techniques, which navigate the source code using software dependencies; and of course hybrid ones, which combine different information sources.

Biggerstaff et al. [Biggerstaff et al. 1994] introduced the problem of concept assignment in the context of static analysis. They implemented a tool which extracts identifiers from the source code and groups them to support identification of concepts. The simplest and most commonly used text-based static technique relies on searching the source code using regular expression matching tools, such as the Unix utility *grep*. Modern development environments like *Eclipse* and *MS Visual Studio* build many useful add-ons on top of simple pattern matching, including references to class and method names, etc. Similarly, Ratiu et al. [Ratiu and Deissenboeck 2007] use simple matching to map concepts to program elements within their formal framework. A significant improvement over regular expression matching is brought by information retrieval-based [Cleary et al. 2009; Gay et al. 2009; Liu et al. 2007; Marcus et al. 2004; Poshyvanyk et al. 2007; Poshyvanyk et al. 2006b] and natural language processing [Hill et al. 2007; Hill et al. 2009; Shepherd 2007; Shepherd et al. 2007] approaches, which allow more general textual queries and provide ranking of the results to these queries. Recent approaches also utilized independent component analysis [Grant et al. 2008], and Latent Dirichlet Allocation [Lukins et al. 2008] to support concept location. One common limitation of these approaches is redundancy in the search results that could be reduced by inferring additional structure among the search results and grouping them based on common topics. Our work tackles this limitation of existing IR-based concept location techniques, augmenting them with formal concept analysis to automatically cluster search results.

One of the first structural-based static techniques for concept location is the one proposed by Chen et al. [Chen and Rajlich 2000], which is based on the search of an abstract system dependence graph. This approach has been recently extended [Robillard 2005; Robillard 2008] via analysis of dependency topologies to rank elements of interest in source code. Some other methods combine other types of information obtained via static analysis (that is, textual and structural), such as Zhao et al. [Zhao et al. 2006] who proposed a technique which combines information retrieval with branch-reserving call-graph information (i.e., an expansion of the call graph with information on branches and sequential information) to automatically assign features to respective elements in the source code. Gold et al. [Gold et al. 2006] proposed an approach for binding concepts with overlapping boundaries to the source code which is formulated as a search problem

using genetic and hill climbing algorithms. A comparison and overview of static feature location techniques can be found in the work of Marcus et al [Marcus et al. 2005b].

Among the techniques that combine static and dynamic information (that is, dependencies and execution traces), of interest is the work of Eisenbarth et al. [Eisenbarth et al. 2003] as it uses FCA to relate features together. FCA is applied to formal contexts consisting of computational units, an executable part of the system (*e.g.* class, module, basic block), as objects and scenarios, a set of user triggered observable events in a system, as attributes, where the relation between objects and attributes identifies computational units exercised by scenarios. Resulting concept lattices identify general and specific computational units as well as relationships between features and computational units. This technique has been recently applied in the industrial setting [van Geet and Demeyer 2009].

FCA has many uses in software engineering [Lienhard et al. 2005; Snelting 2005; Tonella 2003] such as identification of objects in legacy code, however, we discuss here the ones that specifically address concept location. In addition to the work of Eisenbarth et al. [Eisenbarth et al. 2003] (mentioned above), Tonella and Ceccato [Tonella and Ceccato 2004] use dynamic analysis together with FCA to identify aspects in execution traces. Execution traces, obtained by exercising use cases, are objects and executed class methods are attributes of the formal context. Discovery of candidate aspects entail locating use case specific concepts where the intension contains methods from multiple classes and methods of the concept that also appear in the intension of other use case specific concepts. Mens et al. [Mens and Tourwe 2005] apply FCA to mine source code to support various program comprehension tasks, including concept location. In their work objects correspond to classes and methods, and attributes correspond to substrings from class and method names. The resulting concept lattice helps identify design pattern instances, coding and naming conventions, refactoring opportunities and important domain concepts.

Our application of FCA differs from this earlier work. The purpose of FCA, in our case, is to provide automatic clustering and distillation of the search results. We conjecture that inferring and using implicit structure or order in the search results should help reduce the search space and user efforts in terms of locating source code elements related to the concept of interest. We build a formal context, to be used in FCA, by considering source code methods as objects and words (*e.g.*, identifiers and comments) in these methods as attributes.

A comparison of different approaches for feature location in legacy systems is presented by Wilde et al [Wilde et al. 2003]. A more up-to-date summary of all existing approaches can be found in the literature [Antoniol and Guéhéneuc 2006; Revelle and Poshyvanyk 2009], whereas a summary of industrial tools available for feature location is available in the work of Simmons et al [Simmons et al. 2006].

4. CONCEPT LOCATION USING CONCEPT LATTICES

In this section we present the details of our approach to concept location, which uses FCA to organize in a concept lattice the results of a search performed by a developer using the LSI-based source code search engine. Part of the approach is similar to the one presented by Marcus et al. [Marcus et al. 2004] and offers users the same main features, such as, the ability to write queries in natural language and sort the results based on their similarity to the query. With the LSI-based source code search engine, developers search the software much the same way they search the internet with popular search engines like Google. Note that any other text retrieval technique may be used here instead of LSI.

The general process would be the same. The reason we chose LSI is because of our experience with it and the excellent results it produced in previous work.

Fig. 1 shows the main steps in the concept location process using LSI and FCA. The first two steps are usually performed off line (that is, the corpus needs to be built and indexed only after significant changes to the software and it is not interactive), while the other ones are performed interactively and repeatedly during concept location until the user finds the desired parts of the source code.

1. **Creating a corpus of a software system.** The source code is partitioned using a predetermined granularity level (that is, methods or classes) and documents are extracted from the source code. A corpus is created, so that each method (or class) will have a corresponding document in the resulting corpus. Only identifiers and comments are extracted from the source code. We developed tools that automatically create corpora for *MS Visual Studio* C++ projects [Poshyvanyk et al. 2005] and *Eclipse* Java projects [Poshyvanyk et al. 2006a; Savage et al. 2010]. In addition, we also created a corpus builder for large C++ projects, using srcML [Maletic et al. 2002] and Columbus [Ferenc et al. 2004].
2. **Pre-processing and Indexing.** The resulting corpus can be pre-processed using a set of different techniques including stop word removal, splitting identifiers, special token elimination and stemming. The latter is the process of reducing inflected words to their stem or morphological root form (*e.g., depart, departure*, and *departing* are forms of the equivalent lexeme, with *depart* as the morphological root form). Finally, the corpus is indexed using LSI, each document (method or class) has a corresponding vector.
3. **Formulating a query.** A developer selects a set of terms that describe the concept of interest (for example, 'print page'). This set of words constitutes the initial query. The tool spell-checks all the terms from the query using the vocabulary of the source code (generated by LSI). If any word from the query is not present in the vocabulary, then the tool suggests similar words based on editing distance and replaces the term in the search query with the user selected term. It should be noted, however, that in our evaluation of the proposed approach, all the queries were formulated automatically using default feature summaries from user documentation

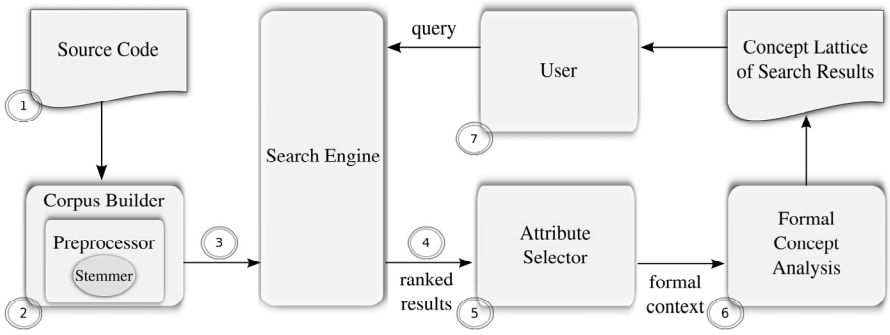

Fig. 1. Concept location process using LSI and FCA. Numbers in the figures correspond to the following steps which are discussed in Section 4 (1) creating a corpus of a software system, (2) pre-processing and indexing, (3) formulating a query, (4) ranking documents, (5) selecting descriptive attributes, (6) applying formal concept analysis, and (7) examining results.

or the short summaries in bug reports. For example, *Mylyn* bug #149838[5] description *"Encoding problems when product name contains foreign characters*" is used as the query. We opted for automatic query formulation technique in order to avoid potential bias in terms of evaluating the results. The query is then represented in the vector space as a pseudo document.

4. **Ranking documents.** Similarities between the user query and the documents from the source code (such as, methods or classes) are computed. The similarity between a query reflecting a concept and a set of data about the source code indexed via LSI allows generating a ranking of documents relevant to the feature. All the documents are ranked by the similarity measure in descending order.
5. **Selecting descriptive attributes.** The top *k* attributes (in the FCA sense) from the first *n* documents in the ranked list (for example, methods) are selected (see Section 4.1). These attributes are the terms that are highly similar to the selection of the *n* documents and less similar to the other documents in the search results.
6. **Applying Formal Concept Analysis.** Before applying FCA we prepare the formal context, which is generated from a set of *n*-first documents (*objects*) in the ranked list and *k* descriptive terms (*attributes*) extracted in the previous step. Subsequently, we apply the FCA algorithm [Ganter and Wille 1996] to build the set of concepts for the given context, which forms a complete partial order, or simply a concept lattice.
7. **Examining results.** The resulting concept lattice, with annotated descriptions for concept nodes and with links to actual documents in source code is presented to the user. The user can browse the results by traversing the lattice and refining queries if desired. If a user finds a part of the concept (that is, the location where a change needs to be done), then the search succeeds, otherwise, the user formulates a new query, taking into account new knowledge obtained from the investigated documents in the lattice, and returns to step 3.

4.1 Selecting descriptive attributes

Steps 1, 2, 3, 4, and 7 from our approach are the same as those used in our previous work [Liu et al. 2007; Marcus et al. 2004; Poshyvanyk et al. 2007; Poshyvanyk and Marcus 2007]. New to the process of concept location are steps 5, 6, and 7, which we describe in more detail here. The addition of the three new steps to the concept location process was inspired by prior work, which utilized FCA to free-text IR-based systems [Cigarran et. al 2004]. We adapt and apply the idea of using FCA for clustering search results, originally introduced by Cigarran et. al. [Cigarran et. al 2004] in order to improve the process of concept location in software.

There are several published solutions to extract descriptive terms for sub-collections of documents. For example, the okapi weighting scheme and the terminological formula are two of the approaches previously used in free-text IR systems [Cigarrán et al. 2004]. We adopt here the technique proposed by Kuhn et al. [Kuhn et al. 2007], since it was defined in the context of source code to select relevant terms with respect to given clusters of source code elements. We present how this technique is adapted and used to select terms to be used in FCA.

We define a corpus for a software system as a set of documents $D = \{d_1, d_2 \ldots d_s\}$. A set of documents in the ranked list which we use to build a formal context is denoted as D_n, where the number of documents is $n=|D_n|$. To denote the rest of the corpus, which does not contain documents in D_n, we use $D^1 = D - D_n$, where the number of documents is $|D^1| = s - n$.

[5] https://bugs.eclipse.org/bugs/show_bug.cgi?id=149838 (*verified on 08/18/2011*)

We define a set of unique terms that occur in D as $T_D = \{t_1, t_2 \ldots t_r\}$. A set of unique terms that occur in D_n only is defined as T_{Dn}, where $T_{Dn} \subseteq T_D$.

In order to rank every term $t_i \in T_{Dn}$ (for i=1 …$|T_{Dn}|$) with respect to a document collection D_n we apply the following formula to determine the ranking of the terms:

$$sim^1(t_i, D_n) = \frac{1}{|D|} \times \sum_{d \in D} sim(t_i, d) - \frac{1}{|D^1|} \times \sum_{d^1 \in D^1} sim(t_i, d^1)$$

The equation above computes the average similarity of a term and all documents in the corpus as well as the average similarity of a term and documents not used to generate the formal context, namely D_n. The difference between the two averages for a given term is returned as $sim^1(t_i, D_n)$. Using this approach we are able to rank all the unique terms in D_n (for example, T_{Dn}) so that the terms highly similar to the documents in D_n but not to the documents in D^1 are ranked higher. We penalize those terms which are highly similar to D^1, since it is observed that they might be identifiers for data structures or utility classes [Kuhn et al. 2007], which would pollute the top ranked list of terms (for example, common function names and keywords, *atoi*, *class*, *sqrt*, etc.).

4.2 Applying formal concept analysis

We decided to use FCA instead of other standard clustering algorithms because of the following reasons: FCA provides an *intensional* description for each cluster, which makes groupings more interpretable; the generated cluster organization is a *lattice*, rather than a hierarchy, allowing easier recovery from bad decisions, while exploring the hierarchy; FCA is generally a more effective way of browsing the document space than hierarchical clustering [Cigarran et al. 2005].

With the approach presented in this paper we tackle the problem of scalability of FCA in the context of a software system by applying it on a *subset* of relevant search results only. Using this approach, the top search results, that is, the first *n* methods or classes in the ranked list are organized in the concept lattice based on the attributes automatically selected from identifiers and comments implemented in their source code. Section 5 discusses in detail the issue of selecting values for *n* and *k*.

4.2.1 *Running Example*. To illustrate how FCA works with respect to the problem that we are addressing in this work, that is, concept location, we present the following example of locating the source code of *Eclipse* 3.1[6] implementing the system functionality which cancels printing after the user initiates a print page request. The following top six methods returned as the results of our initial query *'cancel print page'*: *getBounds()*, which obtains the size of the paper, *startPage()* and *endPage()*, which start and end printing a page, *startJob()*, which initiates a print job which may include printing several pages, *endJob(),* which finalizes printing a page(s), and *cancelJob()*, which ends and cancels the print job respectively.

Using the algorithm for selecting descriptive terms, described in section 4.1, the following terms are selected from the identifiers and comments of the returned methods: *printer*, *print, page, job, device, paper* and *rendering*. Note that these terms are specific only to those six methods and not so much to the rest of the source code in *Eclipse*.

Using the top six methods from the results and their descriptive attributes, we generate a formal context *C = (A, O, R)*, where the objects *O* are the aforementioned methods and *A* are words (*attributes*) extracted from implementation of the methods in *O*. Note that in

[6] http://www.eclipse.org/ (*verified on 08/18/2011*)

Table I. Formal context: objects (six methods from source code of *Eclipse*) and attributes (shared in identifiers and comments of those methods)

	printer	print	page	job	device	paper	rendering
startJob		×		×			
endJob		×		×			
cancelJob		×		×			
startPage			×			×	×
endPage			×			×	×
getBounds	×				×	×	

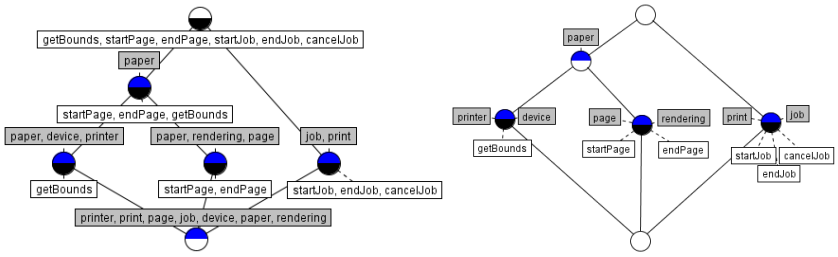

Fig. 2. Concept lattice (left) and its corresponding sparse representation (right) for the '*cancel print page*' query. Grey boxes are attributes (words) and white boxes are objects (methods). The sparse representation of the concept lattice only presents objects (methods) in the smallest concept node which they appear in, eliminating redundant appearances of methods along paths explored by the user.

this example we choose n top objects (n=6) and k most similar terms to these objects (k=7). The set of binary relations R among O and A are summarized in Table I.

When applying FCA on our example, the following concepts are identified:

C_1= ({*startJob*, *endJob*, *cancelJob*}, {print, job})
C_2= ({*startPage*, *endPage*}, {page, paper, rendering})
C_3= ({*getBounds*},{printer, device, paper})
C_4= ({*startPage*, *endPage, getBounds*}, {paper})
C_5= ({*startJob*, *endJob*, *cancelJob, startPage*, *endPage, getBounds*}, {})
C_6= ({}, {*printer, print, page, job, device, paper, render*})

Concepts here are groups of methods (that is, *objects*) and their common words (that is, *attributes*). This set of concepts is referred to as a *complete partial order* whereas some concepts are super- or sub-concepts with respect to others (see Fig. 2). For example, the concept C_4 is a sub-concept of concept C_3. Intuitively, from the intension 'paper' of C_4 we also may assume that C_3 (with intension 'printer device paper') is more specific than concept C_4. The implementations of the methods which belong to these concepts reflect this fact. In addition, the three methods implement different actions related to the paper – *getBounds* is used to obtain physical properties of the paper based on current system device, whereas *startPage* and *endPage* implement operations which initialize and finalize printing of a page respectively.

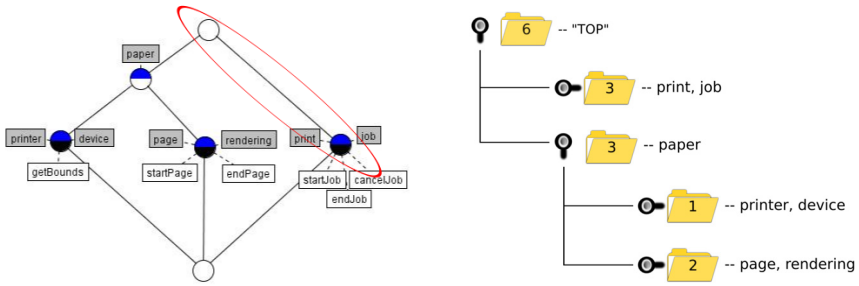

Fig. 3. Concept lattice (left) and tree view (right) for the '*cancel print page*' query. Grey boxes are attributes (words), white boxes are objects (methods), and the path circled in red indicates the *minimal browsing area*. For the tree view each folder represents a concept node and the number on the folder indicates the number of methods that a concept node contains. The labels beside the folder are the terms associated with the concept node.

4.3 Examining results

After applying FCA to a subset of search results, the concept lattice is presented to the user. In order to facilitate the lattice exploration process the user is returned a sparse representation of the concept lattice (see Fig. 2). Presenting the user with a sparse concept lattice eliminates repeated occurrences of methods along a given path in the concept lattice. That is, as opposed to labeling each node with all the elements contained in its intension and extension, attributes (objects) are annotated on a concept node if it is the highest (lowest) node that appears in its intension (extension). For simplicity, we refer to a sparse concept lattice as a concept lattice.

Given a concept lattice, the labels of concept nodes can be viewed by developers to assist them in the navigational decision making process. More specifically, a user should begin evaluating the lattice at the root node. The labels of all sub-concepts should be considered when deciding on the next concept node to visit. Following this decision, all documents of the selected sub-concept node are evaluated. If none of the documents are relevant to the concept of interest, a sub-concept of the current node is selected as previously discussed. The process continues until the developer locates a concept node containing a relevant document. Throughout this process we make an assumption that the attribute labels provide information useful for making navigational decision during concept location.

Consider the example previously discussed in section 4.2.1 where the user is interested in locating methods relevant to the *'cancel print page'* feature. The concept lattice, which appears in Fig. 3, is provided to the user. The exploration of the concept lattice begins at the root node. The attribute labels of all sub-concepts of the root node are considered when making the decision of which node to consider next. In this particular example two concept nodes are considered where the attribute labels are {paper} and {print, job}. Based on these choices the developer might select the concept, which is labeled as {print, job}, as it may be considered to be more relevant to the search query *'cancel print page'*. Following this decision the methods of the concept node, which consist of {startJob, cancelJob, endJob}, are evaluated to determine if a relevant document appears in the concept node. In this particular scenario, evaluation of the selected concept node results in identifying the relevant method *cancelJob* (implements functionality related to canceling a print request) while only having to consider three documents. So during the navigation process, each decision is determined by considering the documents, which

appear in the current concept node, as well as the attribute labels of all sub-concept nodes.

5. EVALUATION OF THE PROPOSED APPROACH

Our approach depends on three specific issues that potentially affect the results: corpus creation (that is, granularity), pre-processing (that is stemming), and applying FCA (that is a number of objects and attributes used). We performed a case study to evaluate our approach and answer the following research questions:

- What is the effect of *n* and *k* when applying FCA and building concept lattices from subsets of search results on the number of methods encountered before locating the first relevant method during concept location?
- What is the effect of stemming the corpus of software on the concept lattices' ability to effectively organize search results in order to limit the number of methods explored by a user during concept analysis?

Our choice of empirical evaluation is based on reenacting concept location based on past changes and to simulate the user's actions. Past changes in software provide us with a change request (or bug description) and the actual changes in the code done in response to the request, named as the change set. During concept location a user or a tool starts with the change request and finds a place in the code where a change should be made. In order to verify that this location is correct, the complete change should be implemented and tested. Reenactment based on historical data allows us to assess the correctness of concept location without the complete implementation and testing. If concept location results in a place in the code that is in the original change set, then we can conclude that concept location succeeded. If the result of the concept location leads to a place that is not in the change set, then we consider that concept location failed. Changes to software can be made in a variety of ways, so there may be some cases when concept location leads to a place that is not in the original changes set, yet could still lead to a complete and correct change. Our assumption will cause to miss these cases, but it is a trade-off we are willing to take given that we gain huge amounts of time in the evaluation.

Many concept location techniques, such as the one we introduced here, depend on user choices. In all cases, it is the user who makes the final decision that a place in the code needs to be changed. In addition, there are other steps in the process where user input and decision is needed. In our approach, two such steps are the most important:

- The query formulation and reformulation (when needed); and
- Navigation of the results.

Since we aim to simulate the user, we have to address these two issues. In order to simulate query formulation during concept location reenactment, we choose as query the original change request (or bug description). We assume that concept location is performed without reformulation of the query, hence no need for further user input. In order to simulate navigation, we assume that users would investigate every piece of code the concept location tool provides in the order it is being provided by the tool. In this way we simulate an "ideal" user behavior, where a single query is formulated without user interference and the results are inspected in the most efficient way. Clearly in real world scenarios concept location is more complex. However, our assumptions allow us to automate in part the evaluation and thus to collect a large number of data points.

Based on our choices and assumptions, the empirical evaluation consists of a case study where our FCA based approach is used to automatically (i.e., simulated user) perform concept location associated with past changes (i.e., reenactment of concept location). A baseline approach, namely, IR-based concept location [Marcus et al. 2004], is also used in a similar fashion and the results are compared to assess whether our

approach can lead to better results (given that both approaches approximate ideal user behavior).

The remainder of this section provides the details of our empirical evaluation: the design of the case study, the data we used, the evaluation mechanism, and the results we obtained.

5.1 Methodology

We evaluated the proposed technique in the case study with a total of 320 features from six open source systems (see Table II). The complete details on all the features and bugs used in the case study are supplied in an online appendix[7]. The online appendix also includes the information on the methods that were changed in order to fix the bugs or implemented to introduce the features (these are extracted from the official patches released to fix the bugs and change history of added or modified features). Additionally, the table in the online appendix also shows method ranks in the list of the results obtained with the baseline approach. As explained above, the search queries to locate the bugs and features were formulated automatically from the descriptions of the bug reports or features from the available user manuals, thus eliminating any potential bias caused by formulating queries by users.

We studied how the number of documents (that is, methods, n) and of terms (that is, attributes, k) affects the size and quality of concept lattices. We also investigated whether stemming of the corpus and queries has an impact on the search results. After an initial study [Poshyvanyk and Marcus 2007] of different lattice configurations with documents and attributes, we decided to keep the number of attributes in the range from 10 to 25 and study the generated concept lattices for the top 80 to 100 documents from the ranked list. Using less than 10 attributes resulted in low clustering capacity in grouping related concepts, while using more than 25 attributes, generated a relatively large number of concept nodes in the lattices making them difficult to navigate.

For every concept location task, given a query and a ranked list of the results, we build 24 concept lattices of different configurations: 12 lattices using all possible combinations of documents (that is, 80, 90, and 100) and terms (that is, 10, 15, 20, and 25), which are obtained from the corpus without applying stemming and 12 lattices with the same document-term configurations for the stemmed corpus. In order to perform a fair comparison between concept lattices with varying number of documents and with or without stemming, we only considered queries where the first relevant method appeared in the top 80 for both the stemmed and non-stemmed scenario. Such criteria ensured that metric values for the concept lattices are defined. We derive our analysis results from the concept lattices created for the features of *ArgoUML*, *Freenet, iBatis, JMeter, Mylyn and Rhino* (see Table II for the number of concepts located per system).

5.2 Design of the case study

The case study design is based on the guidelines defined by Yin [Yin 2003]. The results of the proposed approach based on the combination of FCA and LSI with stemming are compared against the results that do not utilize stemming. Both configurations of this combined approach are compared against an LSI-based ranking of the results (that is, the *baseline approach*).

[7] http://www.cs.wm.edu/semeru/tosem-fca-lsi (*verified on 08/18/2011*)

5.2.1 *Objectives*. The goal of the case study is to evaluate the impact of the following parameters and pre-processing technique on the size and quality of resulting concept lattices:

- the number of documents (*n)* in the ranked list that should be kept for selection of descriptive attributes and the final concept lattice;
- the number of attributes (*k)* that should be selected for the number of *n* documents;
- the stemming of the corpus and the queries.

We expect that the resulting concept lattices will help reducing developer searching efforts when compared to the IR-based concept location (baseline) technique, which does not cluster the results. This hypothesis is based on the fact that the new approach can effectively utilize information about relationships among the results of the search based on common attributes rather than only those used in the original user query. In other words, it can effectively group relevant documents and provide informative labels as node descriptions in a concept lattice, helping users to navigate the resulting lattice more effectively, possibly navigating through only fraction of the documents. Such a representation should provide a structured view of different sub-topics present in the results of the search and provide additional information, such as descriptive labels, which can be used as visual cues to navigate results more effectively than a plain ranked list. This hypothesis, however, can be investigated via user studies, which we are planning to conduct in our future work.

With respect to applying stemming we conjecture that our approach might not only benefit in the quality of the terms selected by the attribute selector, but also in the resulting clustering of the topics. For instance, consider the case where the terms 'table' and 'tables' both appear in the set of relevant attributes in a subset of search results. Ideally, each selected term should assist in identifying unique topics in the set of methods. In this example, generating a concept lattice using both terms would not seem to provide better reduction of search effort as compared to lattices using only one of these attributes. In the stemmed version of the same corpus the term 'tabl' would appear as a representative of both 'table' and 'tables' allowing for the selection of another, potentially more useful term, thus helping reduce effort required to locate relevant information in the subset of search results.

5.2.2 *Objects and settings of the case study*. We utilized the following software systems in our case study: *ArgoUML*[8] (version 2.8), *Freenet*[9] (version 0.7), *iBatis*[10] (version 2.3), *JMeter*[11] (version 2.3.4), *Mylyn*[12] (version 1.0.1), and *Rhino*[13] (version 1.5 release 6). *ArgoUML* is an open source java implementation of a UML diagramming tool. *Freenet* is an open source implementation of a peer-to-peer anonymous file sharing software. *iBatis* is an object-relational mapping tool that facilitates the mapping of SQL databases to objects in a variety of programming languages. *JMeter* is an open-source Java desktop application developed to allow users to load test functional behavior of web applications and other functions. *Mylyn* [Kersten and Murphy 2006] is a well-known Eclipse plug-in which facilitates task oriented development. Our evaluation is performed using only two components from *Mylyn*, *bugzilla.core* and *bugzilla.ui*, for which the mappings between bugs and source code were made publicly available [Kersten and

[8] *ArgoUML*: http://argouml.tigris.org/ (*verified on 08/18/2011*)
[9] *Freenet*: http://freenetproject.org/ (*verified on 08/18/2011*)
[10] *iBatis*: http://*ibatis*.apache.org/ (*verified on 08/18/2011*)
[11] *JMeter*: http://jakarta.apache.org/jmeter/ (*verified on 08/18/2011*)
[12] *Mylyn*: http://www.eclipse.org/mylyn/ (*verified on 08/18/2011*)
[13] *Rhino*: http://www.mozilla.org/rhino/ (*verified on 08/18/2011*)

Murphy 2006]. *Rhino* is an open-source software system that provides a Java implementation of JavaScript.

We preprocessed and indexed the source code of each system using the approach outlined in Section 4. We chose method level granularity, *i.e.*, each document in the corpus corresponds to a method. We constructed the corpus for each system by extracting all comments and identifiers from the source code. The resulting text was pre-processed using the following set of rules: some types of tokens were eliminated (for example, operators, special symbols, some numerals, keywords of the Java programming language, standard library function names); the identifiers in the source code were split into parts based on known coding standards while the initial form of each identifier was preserved as well; each document in the corpus was created with the comments and identifiers corresponding to each method. Note that a comment is associated with a method if it appears immediately before, within the body, or immediately after a method definition. In our future work, we will consider applying more advanced approaches for associating comments with methods in source code like the ones proposed by Fluri et al. [Fluri et al. 2009]. No morphological analysis or transformations were applied since we did not use a pre-defined vocabulary, or a pre-defined grammar. In our case studies we also utilized stemming, using the widely adopted Porter stemmer algorithm[14], and evaluated its impact on the proposed technique. The size of the corpora and the number of indexed methods (that is, parsed documents) from each system are outlined in Table II.

5.2.3 *Data used.* As mentioned before, one recurrent issue in empirical studies on concept location is the verification of the results. It is often difficult to determine for sure that a certain method implements, at least in part, a given concept. To mitigate this problem, we utilize datasets obtained using three complementary strategies for establishing mappings between source code elements and concepts.

The first strategy validates the existence of a correspondence between a concept expressed in a change request and source code elements by inspecting the actual history of changes behind a given change request. Of course, a given change request may be designed and implemented in many ways. In order to minimize the threats to the validity of our results, we opted to employ the first strategy similarly to its previous usage [Cleary et al. 2009; Lukins et al. 2008; Poshyvanyk et al. 2007; Poshyvanyk and Marcus 2007]. Specifically, we reenact concept location associated with previous changes associated with particular bugs reported for the given software. During reenactment developers perform concept location starting from the bug description and we can verify the correctness of the location process by checking the final patches for these bugs, as those are available and are not implemented by any of the authors of this paper. The documentation for every bug used in the case study specifies which methods were changed during the bug fix. We consider these methods as (part of) the implementation of the concept associated with the bug. We used the following criteria to select bugs for the case

Table II. Software systems' source code and corpus vitals

System	# of Features	LOC	# Parsed Docs	Vocabulary (stemmed)	Vocabulary (non-stemmed)
ArgoUML	27	308K	10,546	2,803	10,459
Freenet	33	295K	18,147	6,438	17,295
iBatis	13	13K	1,869	684	2,041
JMeter	85	130K	8,269	2,699	8,489
Mylyn	57	13K	537	668	1,551
Rhino	105	32K	3,800	1,989	5,326

[14] *Porter Stemmer:* http://tartarus.org/~martin/PorterStemmer/ (*verified on 08/18/2011*)

study: (1) bugs should be well-documented and reproducible; (2) bugs should have approved patches applied in recent releases; (3) none of the authors knows the parts of the program corresponding to the features to eliminate potential bias; (4) a short description of a bug report is available and can be used as a query input to an IR-based source code search engine to eliminate potential bias while formulating queries. Short descriptions of bug reports typically describe the problematic feature. For each bug we are interested in using the short description of the bug report as a query and locating at least one of the methods modified during its fix and found in the official patch. For our evaluation, the subset of the *Mylyn* dataset which was established using bug reports and previously used by Eaddy et al. [Eaddy et al. 2008b], follows such criteria to map concepts to source code.

The second strategy in our evaluation uses the prune dependency rule introduced by Eaddy et al. [Eaddy et al. 2008b] to establish mappings between source code elements and concepts. Eaddy et al. [Eaddy et al. 2008b] released a set of mappings for a number of concepts derived from the software system documentation and source code elements for a set of open source software systems. Our case study utilizes this publicly available data[15] that has been previously published and verified by other researchers [Eaddy et al. 2008a; Eaddy et al. 2008b]. Moreover, in the second strategy, official system documentation is used to derive the search queries (that is, relevant sections of documentation are used as queries) used in our evaluation. The datasets that use the second criteria include *iBatis*, *Mylyn*, and *Rhino*.

Finally, the third strategy used to establish mappings between source code elements and concepts entails researchers manually identifying such relationships. Our case study also utilizes publicly available benchmarks[16], which have been used in previous studies [Bacchelli et al. 2010a; Bacchelli et al. 2009; Bacchelli et al. 2010b]. The creators of the benchmarks linked a set of emails from development mailing lists to source code elements. Six research group members manually mapped the emails to source code with two researchers reviewing 51% of the emails. In the context of our case study, the text within the email discusses concepts implemented in the source code. Therefore, we used the emails as queries corresponding to the concepts that we were interested in locating in the source code. Software systems from the benchmarks discussed above which are used in our evaluation include *ArgoUML*, *Freenet*, and *JMeter*.

5.2.4 *Evaluation criteria and measures*. We compare the results of our new concept location technique with the sorted list of the retrieval results obtained with the IR-based ranking of the source code elements (that is, the baseline approach). We assume that with a ranked list, a user has to scan documents (that is, methods), starting from the first one, until the relevant document is found. We define the scope of concept location to finding the starting point of a change, in our case fixing a bug or implementing a concept, as defined by Rajlich and Gosavi [Rajlich and Gosavi 2004], as it is the role of impact analysis and change propagation to get the full extent of the change in the source code [Rajlich and Gosavi 2004; Ren et al. 2004]. A study on how developers explore search results suggests that developers actually "skim" results as opposed to performing exploration systematically [Starke et al. 2009]. However, in this work we assume that developers explore results systematically based on the technique they use, to allow for stability during empirical comparison of techniques. At the same time we understand that the same comparison involving human subjects could produce different results.

It is common in retrieval tasks to use *precision* and *recall* [Baeza-Yates and Ribeiro-Neto 1999] as measures to assess the quality of the retrieved results.

---

[15] http://www1.cs.columbia.edu/~eaddy/concerntagger/ (*verified on 08/18/2011*)
[16] http://miler.inf.usi.ch/ (*verified on 08/18/2011*)

Precision and *Recall* are two well known Information Retrieval metric and are defined as follows:

$$Precision = \frac{|correct \cap retrieved|}{|retrieved|} \qquad Recall = \frac{|correct \cap retrieved|}{|correct|}$$

where *correct* and *retrieved* correspond to the set of *relevant* documents and the entire set of *retrieved*, respectively.

Precision and recall are complementary measures and usually increasing one of them results in the decrease of the other. Depending on the application (i.e., retrieval tasks), high precision, high recall, or acceptable balance of the two is desired. For example, during concept location, high precision is desirable as the developer wants to investigate as few false positives as possible. On the other hand, during impact analysis, for example, high recall is more important as it is undesirable to miss places in the code that need to be changed.

Concept location techniques that are formulated as retrieval tasks, as in this paper, are evaluated in related work using these two measures, adapted for the specifics of the task at hand [Poshyvanyk et al. 2007]. It is important to note again that concept location succeeds when one of the methods that need to change is identified. In this context, recall becomes one (1) when concept location succeeds and zero (0) otherwise. In consequence, the measure best suited to compare the performance of concept location techniques is precision when recall is one (i.e., when concept location succeeds). In such situation precision is computed simply as the inverse of the number or documents (i.e., methods in our case) that are investigated (i.e., retrieved) until the proper one is found (see section *Maximum possible precision gain*). In order to make the precision relevant to developer actions, it is common to use the inverse precision, when comparing concept location techniques. Previous work defined this measure as *effectiveness* [Poshyvanyk et al. 2007] and it is used to approximate user effort, as it represents the number a methods the user needs to investigate in order to conclude concept location. Since our focus is on the number of methods scanned to reach the best ranked relevant method, relevant methods with the same rank do not impact our *effectiveness* measure. In reality, users may spend as little as a few seconds looking at a method (or only to its signature) or considerable amount of time. To simplify the evaluation we consider that each method inspection takes the same time, that is, we do not distinguish between the methods they inspect. Again, in order to account for such a factor, user studies are needed.

In the case studies we use the inverse precision at maximum recall (i.e., one), a.k.a. effectiveness, to measure and compare the performance of the two concept location techniques. As mentioned before, we simulate user actions, so we need to define how the "users" investigate the results in order to measure precision and recall, and implicitly the effectiveness. For the baseline approach, we assume that the ideal "user" would investigate every method in the ranked list of results until the target one is reached. For the FCA-based approach, we assume that the ideal "user" will navigate the concept lattice starting at the root and investigating every method in the nodes encountered on the shortest path from the root to the target method. This approach is not uncommon, the shortest path was previously used to define the *minimal browsing area* [Cigarrán et al. 2004], which is in fact the measure we use.

In our context, the *minimal browsing area* and the *effectiveness*, described above, are formally defined as follows.

Minimal browsing area

Let C be the set of nodes in the resulting concept lattice. The programmer, while visiting a node in the lattice, can view the actual objects, which correspond to this node (that is, methods from the software system). We define $C_{FEATURE} \subseteq C$ as the subset of the concept nodes containing methods relevant to the feature, which are present in the concept lattice. We redefine the *minimal browsing area* (MBA) as the minimal part of the lattice that a user should explore, starting from the top node, to reach the first object in $C_{FEATURE}$. The reason for redefining this metric is to allow its definition to reflect the task of concept location.

Given two concept nodes c_i and c_j, the node c_i is smaller than c_j if the extension of c_i is a subset of the extension of c_j. Given a document $d \in D$, its *object concept* is the smallest concept in C which contains d in its extension. Within a concept lattice a path, p, from the root node to a given node in the concept lattice can be expressed as a sequence of concept nodes $c_0, c_1,\ldots,c_n$ where $n \geq 0$ such that each c_i is a concept node and c_{i+1} is a sub-concept of c_i. For each concept node in $C_{FEATURE}$ there exists at least one path from the root to each concept node, which appears in the set. We define $P_{FEATURE}$ as the set of all paths from the root concept node to concepts of $C_{FEATURE}$. In order to measure the effort required to traverse a particular path, for each concept node contained in the path we count the number of documents, d, with its object concept appearing in $C_{FEATURE}$. More specifically, for each concept node in $C_{FEATURE}$ we count the number of documents which appear in the concept node, but do not appear in any of the sub-concept nodes. Note that we provide the user with a sparse representation of the concept lattice, which, for each concept node, only indicates documents that meet the aforementioned criteria. MBA is identified by selecting the path from $P_{FEATURE,}$ which requires minimal effort to traverse. That is, locating the path which requires the user to investigate the minimal number of methods (i.e., the lowest possible number of false positives) while navigating to a method of interest. Determining MBA can easily be transformed into the problem of finding the shortest path between the root node and a node in $C_{FEATURE}$.

Effectiveness

We need a uniform measure to compare the effectiveness of the proposed and the baseline concept location techniques. More specifically, our proposed concept location technique returns a concept lattice that developers traverse as opposed to a ranked list, which is traditionally used to compute precision and recall. Since the goal of every feature location technique is to reduce the effort for the developers in the concept location process, we measure this effort as the number of methods from the list that the developers would need to investigate until they find the first relevant method. Formally, we define effectiveness of a baseline technique, that is E_{RL}, and the techniques utilizing FCA, that is E_{MBA}, as the rank $r(m_i)$ of the methods m_i, where m_i is the top ranked method among the methods that must be changed. The effectiveness captures how many methods must be investigated before the first method relevant to the feature is located. A higher effectiveness value indicates that more search effort is needed. Using the same example in Fig. 2, the method of interest occurs in the baseline approach in position 6, having $E_{RL} = 6$. However, in the concept lattice it is in position 3, thus $E_{MBA} = 3$. As previously mentioned the MBA reflects the best possible performance for a given concept lattice. It is possible to achieve lower effectiveness if the user does not navigate the minimal browsing area.

Maximum possible precision gain

Finally, we must compare the performance of the baseline technique with the proposed technique. Here we define *maximum possible precision gain* (MPG), a measure which extends the commonly used *precision* [Baeza-Yates and Ribeiro-Neto 1999] measure to capture the potential gain provided by our concept lattice based technique. The reason we are talking here about a *maximum possible gain* in precision is because using the minimal browsing area (i.e., the methods along the shortest path) in the measure of precision we compute the precision in the best possible case (i.e., assume an ideal user). In real life scenarios, users may take different paths in the lattice. Nonetheless, MPG shows us whether in the best case scenario, the FCA based technique leads to better precision than the baseline or not.

P_{MBA}, is defined as the number of relevant documents encountered (the document representing the first relevant method) divided by the total number of irrelevant documents which viewed when traversing the minimal browsing area. Obviously, the lower bound is the size of the ranked list of search results that the user has to scan before he identifies the first method belonging to the feature. We denote the precision of the ranked list as P_{RL}. Formally the measures P_{MBA} and P_{RL} are defined as follows:

$$P_{MBA} = \frac{1}{E_{MBA}} \qquad P_{RL} = \frac{1}{E_{RL}}$$

Notice that the numerator is constant, with a value of one, because we are interested in locating only one method, namely the first relevant method, as we explained before.

We define the *maximum possible precision gain* as the utmost possible precision gain obtained with the concept lattice over the precision of the ranked list.

$$MPG(C) = \frac{P_{MBA} - P_{RL}}{P_{RL}} \times 100\,\%$$

Consider the example from Fig. 2 (see Section 4.2.1) and let us assume that the developer is locating the method that cancels printing operations and the method of interest occurs in position 6 (that is *cancelJob*) out of 7, having P_{RL}=0.16. However, in the concept lattice it has the ranking of 3 out of 7, thus P_{MBA}=0.33. To obtain the ranking of a document in a concept lattice we count the number of documents, which appear in the concept nodes along the path to the document of interest. Eventually, MPG(C) = (0.33-0.16)/0.16 = 106%, meaning that the concept lattice can reduce the effort in half when compared to a plain ranked list in this particular case.

Positive values of MPG indicate a potential gain in performance when the concept lattice based approach (and the user follows the MBA) is applied as opposed to the use of the IR-based ranked list for concept location. When MPG returns a value of zero, the technique would encounter an identical number of irrelevant documents before locating the first relevant document. Note that it is possible for the metric to yield a negative value, which indicates that the precision obtained using the IR-based ranked list is an improvement over the results acquired using the concept lattice based approach. Additionally, as previously mentioned, MPG is based on the precision of MBA, therefore providing an upper bound to the gain in precision possible when using concept lattice based concept location. In practice, such gains are only possible when navigation decisions made by developers during lattice exploration are identical to the traversal path, which minimizes the effort required to locate the first relevant method, namely the MBA. We cannot guarantee that a user will always follow such a path during evaluation of concept lattices. Therefore, the proposed metric serves as a valid indicator of the

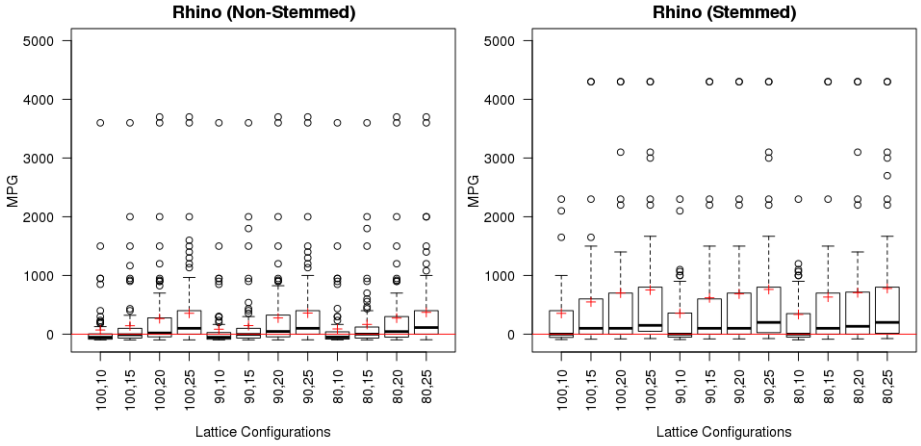

Fig. 4. Box-plots for the descriptive statistics for MPG values for different configurations (documents - *N* and terms - *K*) of concept lattices of *Rhino* for both stemmed (right) and non-stemmed (left) version of the corpora. For each box plot, the bottom and top whiskers represent the minimum and maximum non-outlier values respectively, the bottom and top of the box represent the 1st and 3rd quartiles respectively, and the line inside the box represents the median, whereas the crosses inside the boxes indicate the mean. Circles represent data points determined to be outliers (1.5*(3rd Quartile - 1st Quartile)).

potential gain in precision, which can be acquired through the use of the concept lattice in the best possible exploration scenario.

5.3 Results and discussion

Fig. 4 through Fig. 9 provide the descriptive statistics for the MPG values for each lattice configuration across all the features in six software systems. We provide the mean, 1st quartile, 3rd quartile, and maximum values for both non-stemmed and stemmed versions of the corpora. We differentiate the descriptive statistics among various configurations of lattices to distinguish general and software specific trends. MPG expresses potential precision gain acquired by using concept lattices as compared to LSI-based ranked lists of the results. Negative values indicate scenarios where LSI-based concept location outperforms the proposed technique.

5.3.1 *The results for Rhino.* The results of the study indicate that *the proposed approach outperforms the IR-based concept location technique for all the configurations, with the best results obtained while increasing the number of terms and decreasing the number of documents*. The results for locating 105 features in *Rhino* (see Fig. 4) using 2,520 concept lattices indicate that the MPG values range from -85.1% (-93% for stemmed) to 7,700% (7,900% for stemmed) depending on specific configuration of a concept lattice. The results indicate that there are cases where the concept lattices significantly outperform the IR-based ranked lists and vice versa, shown by the range (that is min to max) of MPG for various configurations. The mean MPG values across all the configurations indicate that the proposed approach outperforms the IR-based ranking of the results. In this case mean values of MPG are positive for all configurations, indicating that, on average, there is a reduction in the number of methods evaluated to locate the first relevant method. Moreover, studying the average MPG values per configuration unveils the existence of a positive correlation between a number of terms used to build lattices and the MPG values. The configurations using 100 documents indicate a consistent increase in the MPG values from 375% when 10 terms are used up

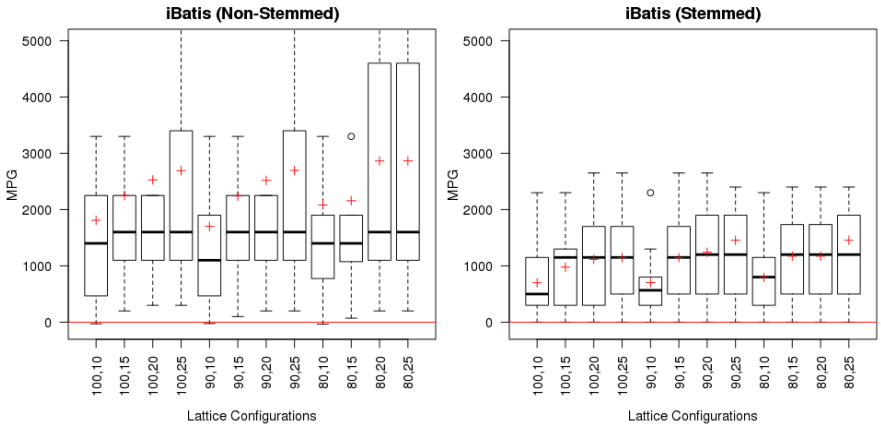

Fig. 5. Box-plots for the descriptive statistics for MPG values for different configurations (documents - *N* and terms - *K*) of concept lattices of *iBatis* for both stemmed (right) and non-stemmed (left) version of the corpora. For each box plot, the bottom and top whiskers represent the minimum and maximum non-outlier values respectively, the bottom and top of the box represent the 1st and 3rd quartiles respectively, and the line inside the box represents the median, whereas the crosses inside the boxes indicate the mean. Circles represent data points determined to be outliers (1.5*(3rd Quartile - 1st Quartile)).

to 1,383% when 25 terms are used. This trend holds for all the other configurations using 80 and 90 documents.

While increasing the number of terms from 10 to 15 the upper 75% of the data points own positive MPG values indicating that the proposed approach outperforms the IR-based concept location approach in at least 75% of the cases. The results also support the observation that decreasing the number of documents leads to increased MPG values. It should be noted, however, that this improvement is not as significant and consistent as the improvement caused by increasing the number of terms. There are a few scenarios where decreasing the number of documents negatively impacts the MPG values. For instance, consider the configurations n=90 and k=20, and n=80 and k=20 where MPG decreases from 1,064.8% to 1,012.8% (in average). However, even in those cases the concept location using concept lattices outperforms the IR-based concept location approach. Moreover, the results of statistical Wilcoxon's matched-pairs signed-ranked test with an alpha of 0.05 for comparing differences in effectiveness for both approaches, E_{RL}-E_{MBA}, indicate that the majority of lattice configurations yields statistically significant improvement in effectiveness while using concept location technique with concept lattices (see Table III).

Concept location with the concept lattices technique using corpus stemming outperforms its carbon copy, which does not utilize stemming. Our findings for *Rhino* advocate that applying stemming enhances the performance of the proposed technique to efficiently locate concepts in source code. For example, stemming prompts an elevation in the MPG values for the configuration using 80 documents and 25 terms (1,514.9% without stemming vs. 2,812.5% with stemming). Although cases exist where MPG has a few negative values, as indicated by the minimal values column, the upper 75% of the data for each configuration include only positive MPG values.

5.3.2 *The results for iBatis*. The results for *iBatis* (see Fig. 5) indicate that the proposed technique outperforms the IR-based concept location approach even without utilizing corpus stemming. In this particular case we have 13 features from which we

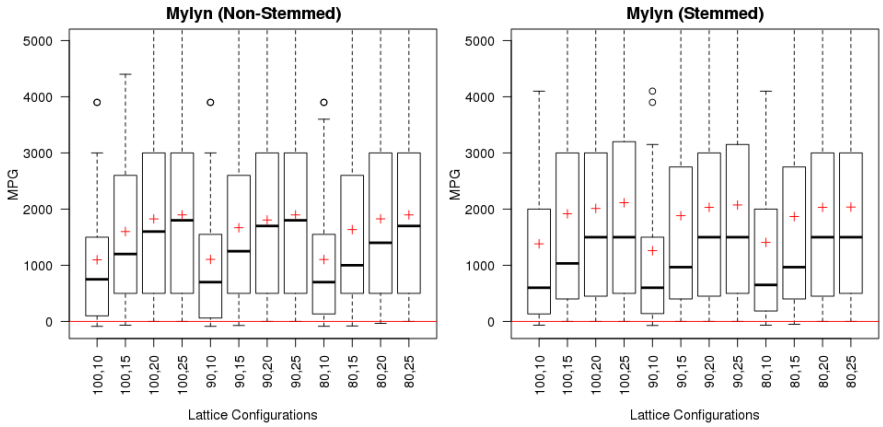

Fig. 6. Box-plots for the descriptive statistics for MPG values for different configurations (documents - *N* and terms - *K*) of concept lattices of *Mylyn* for both stemmed (right) and non-stemmed (left) version of the corpora. For each box plot, the bottom and top whiskers represent the minimum and maximum non-outlier values respectively, the bottom and top of the box represent the 1st and 3rd quartiles respectively, and the line inside the box represents the median, whereas the crosses inside the boxes indicate the mean. Circles represent data points determined to be outliers (1.5*(3rd Quartile - 1st Quartile)).

generate 312 lattices to evaluate. The MPG ranges from -33.3% (*n*=80 and *k*=25) to 7,100% (all configurations). All the configurations consisting of more than ten terms boost MPG values. Any concept lattice corresponding to these configurations outperforms the baseline approach. While using concept lattices, applying stemming does not result in any significant improvement. The results of the IR-based concept location technique improved significantly after applying stemming, thus, leaving little or no room for improvement for the proposed concept lattices based approach. That is, improvement of the IR-based concept location technique allows the position of the first relevant method to near or reach one, which is considered optimal. A baseline method, which yields results close or equal to the optimal result limits or eliminates any possible improvements. If the first relevant method acquired using the baseline is located in the first position, alternate techniques can only equal the performance at best, as it is practically impossible to exceed such a result. For the stemmed version of the corpus, the MPG values range from 0% (all configurations) to 5,400% (*n*=90 and *k*=25).

5.3.3 *The results for Mylyn*. As visualized in Fig. 6, for nearly 85% of the configurations obtained using the non-stemmed version of the corpus and 90% of the configurations of the stemmed version of the corpus the minimum value of MPG is zero. The observation is made using the total 1,368 concept lattices generated for the various configurations of the 57 features of *Mylyn* evaluated. Such a result is significant as it indicates that even in the worst case the performance of the proposed concept location technique exceeds that of the baseline approach for those particular configurations. Overall all configurations considered, the maximum possible precision gain ranges from -86% (-71% for stemmed) to 6,200% (7,000 for stemmed). Once again, analyzing the mean MPG value across the various configurations shows the existence of a positive correlation between the number of terms used and the improvement obtained through the use of concept lattice based concept location.

5.3.4 *The results for ArgoUML*. Our finding for *ArgoUML* show a slight decrease in the average MPG values for the 648 lattice configurations evaluated corresponding to the selected 27 features (see Fig. 7). The average MPG values range from 167% (212% for

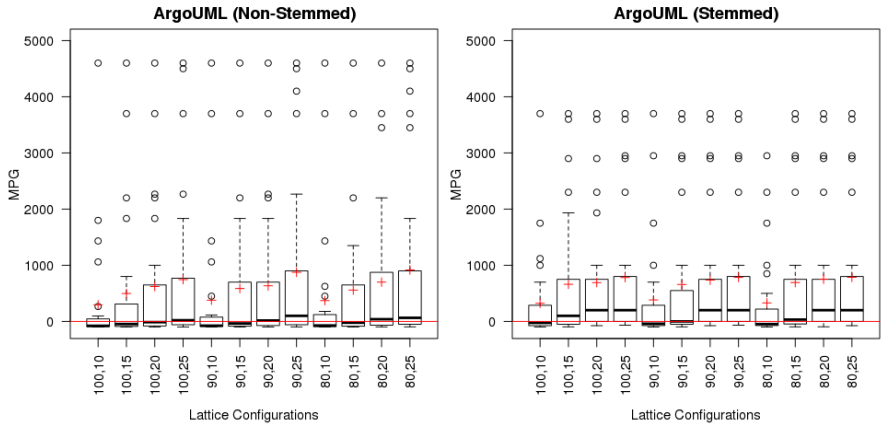

Fig. 7. Box-plots for the descriptive statistics for MPG values for different configurations (documents - *N* and terms - *K*) of concept lattices of *Mylyn* for both stemmed (right) and non-stemmed (left) version of the corpora. For each box plot, the bottom and top whiskers represent the minimum and maximum non-outlier values respectively, the bottom and top of the box represent the 1st and 3rd quartiles respectively, and the line inside the box represents the median, whereas the crosses inside the boxes indicate the mean. Circles represent data points determined to be outliers (1.5*(3rd Quartile - 1st Quartile)).

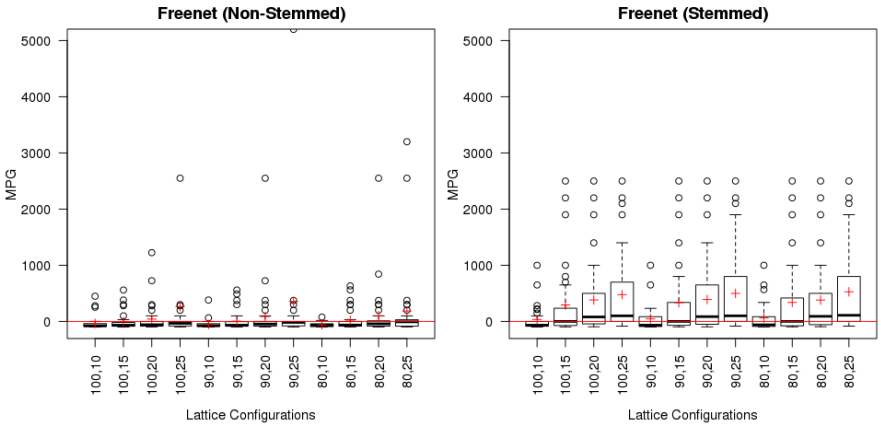

Fig. 8. Box-plots for the descriptive statistics for MPG values for different configurations (documents - *N* and terms - *K*) of concept lattices of *Mylyn* for both stemmed (right) and non-stemmed (left) version of the corpora. For each box plot, the bottom and top whiskers represent the minimum and maximum non-outlier values respectively, the bottom and top of the box represent the 1st and 3rd quartiles respectively, and the line inside the box represents the median, whereas the crosses inside the boxes indicate the mean. Circles represent data points determined to be outliers (1.5*(3rd Quartile - 1st Quartile)).

stemmed) to 502% (792% for stemmed). Although the magnitude of improvement acquired when applying our concept lattice based technique on *ArgoUML* is not comparable to the results of the systems discussed above the general trend of the data is virtually identical. Once again, the proposed technique benefits from the use of stemming.

5.3.5 *The results for Freenet.* The results of *Freenet* present an unusual case when compared to other systems in the case study (see Fig. 8). For this data set (33 features which resulted in 792 concept lattices) we encounter configurations where, on average, the baseline approach outperforms our proposed technique without stemming with

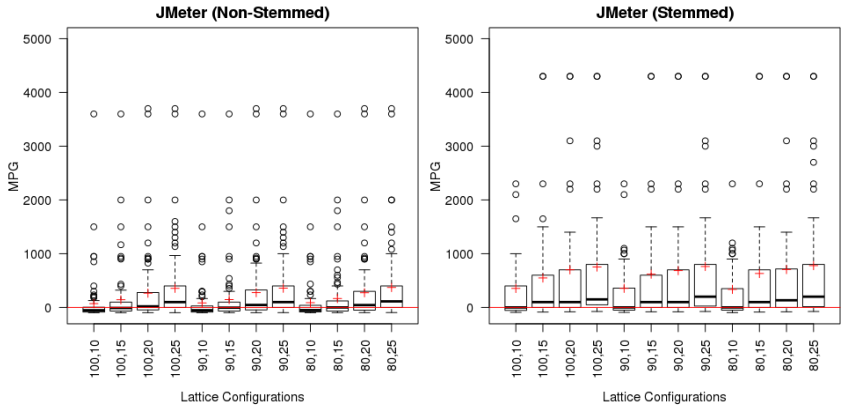

Fig. 9. Box-plots for the descriptive statistics for MPG values for different configurations (documents - *N* and terms - *K*) of concept lattices for both stemmed (right) and non-stemmed (left) version of the corpora. For each box plot, the bottom and top whiskers represent the minimum and maximum non-outlier values respectively, the bottom and top of the box represent the 1st and 3rd quartiles respectively, and the line inside the box represents the median, whereas the crosses inside the boxes indicate the mean. Circles represent data points determined to be outliers (1.5*(3rd Quartile - 1st Quartile)).

respect to maximum possible precision gain. For example, the configuration where n=100 and k=10 the average MPG value is -13%. There are two important observations which must be made. First, when stemming is applied the average MPG value for the configurations considered ranges from 17% to 485% as opposed to -24% to 83% without stemming. Such results further highlight the pertinent role that stemming plays on our technique. Second, as the number of terms increase we observe an increase in the MPG values. Based on this observation, which appears across all the systems in this case study, we conjecture that if we were to use larger number of terms we could obtain results similar to those obtained for other software systems evaluated.

5.3.6 *The results for JMeter.* The MPG values achieved range from 62% (368% for stemmed) to 360% (828% for stemmed) for the configurations of the 85 features (2,040 concept lattices) evaluated. As is the case with other systems evaluated, the impact of stemming on the results prevails as shown in Fig. 9. Also, the results for *JMeter* demonstrate the improvement in performance acquired when the number of terms used to generate the concept lattice increases. For example, increasing the number of terms from 10 to 15 when 100 documents are used increases the MPG value from 62% to 131% (391% to 588% for stemmed).

5.4 Analysis of concept lattice configurations

While our analysis of the results of MPG values supports the assumption that concept lattices are effective in terms of clustering source code search results, we also compared lattices of different configurations to gain more insights into the differences in their effectiveness. Fig. 10 provides a visual synopsis of the descriptive statistics, combining results from all software systems, summarizing the differences in the effectiveness while comparing lattices of different configurations to the IR-based ranking of the results. In this case, each data point is the difference between the absolute position of the first relevant method in the concept lattice, which follows the minimal browsing area, and the position of the first relevant method in the ranked list of search results using the baseline approach. Each data point represents the difference in the effectiveness for the two concept location techniques while locating a single feature or a bug, whereas each box

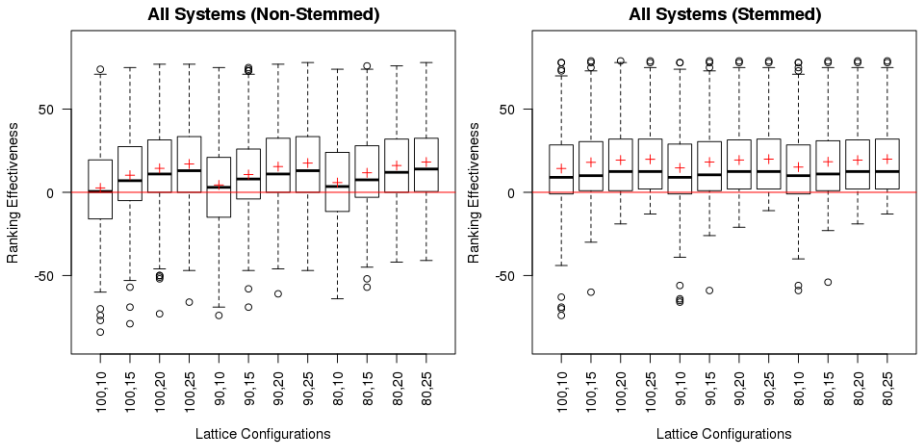

Fig. 10. The box plots summarize the difference in effectiveness, that is, E_{RL}-E_{MBA}. The dataset for each box plot consists of the absolute difference E_{RL}-E_{MBA}. For each box plot, the bottom and top whiskers represent the minimum and maximum non-outlier values respectively, the bottom and top of the box represent the 1st and 3rd quartiles respectively, and the line inside the box represents the median, whereas the crosses inside the boxes indicate the mean. Circles represent data points determined to be outliers (1.5*(3rd Quartile - 1st Quartile))

plot summarizes the descriptive statistics of these differences across all the features and bugs for a given lattice configuration. For instance, the box plot summarizing differences between effectiveness of concept location technique using lattice configurations with n=80 and k=25 over the IR-based concept location approach (see Fig. 10) indicates that concept lattices reduce the number of methods in the search results that need to be inspected by more than 10 methods on average as compared to the baseline approach. This box plot also indicates that there are only a few cases where the IR-based ranking slightly outperforms the concept lattices. The results for locating the features in *Rhino*, *iBatis, Mylyn, ArgoUML, Freenet,* and *JMeter* are summarized in the box plots.

Overall, the box plots indicate the presence of all the trends identified and described in the previous section. For instance, it can be observed that increasing the number of terms elevates the effectiveness of concept lattices as compared to the baseline approach. The case of concept location using non-stemmed version of the corpus provides a clear example of how increasing the number of terms in lattices improves overall effectiveness. While using 100, 90 and 80 documents with 10 terms, it is evident that the IR-based concept location approach outperforms concept lattices in approximately 50% of the cases for each configuration. However, the effectiveness of concept lattices is directly proportional to the number of terms used to build lattices. In the same setting, but using 25 terms, the concept lattices outperform the baseline approach in approximately 75% of cases. This trend also holds for stemmed versions of the corpora.

The results indicate noticeable positive impact of stemming on concept location using lattices. After stemming the corpora, the average of the effectiveness measure for all lattice configurations indicates an improvement over the baseline approach. The box plots confirm that after applying stemming all the lattice configurations outperform the baseline approach in approximately 75% of the cases. The results highlight the significance of choosing the number of terms for building lattices as selecting adequate values for this parameter (*e.g.*, at least 15 terms) can warrant consistent improvements of using concept lattices over the IR-based ranking of the search results.

Finally, the results of statistical Wilcoxon's matched-pairs signed-ranked test with an alpha of 0.05 for comparing differences in effectiveness between the baseline and

Table IV. The results of Wilcoxon's matched-pairs signed-ranked test with an alpha of 0.05 for comparing differences in effectiveness, that is, E_{RL}-E_{MBA}. Highlighted are lattice configurations that provide statistically significant differences while using stemming.

	P-value			
N	K=10	K=15	K=20	K=25
100	0.0010	< 0.0001	< 0.0001	< 0.0001
90	0.0003	< 0.0001	< 0.0001	< 0.0001
80	0.0001	< 0.0001	< 0.0001	< 0.0001

Table III. The results for Wilcoxon's matched-pairs signed-ranked tests for checking if applying stemming yields statistically significant impact on LDF measure. Highlighted are the configurations, which indicate statistically significant difference in the results (in some cases stemming improves the position of the first relevant method in the baseline approach providing little or no room for improvement while using concept lattices).

		P-value			
System	N	k=10	k =15	k =20	k =25
Rhino	100	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	90	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	80	< 0.0001	< 0.0001	< 0.0001	< 0.0001
iBatis	100	0.9768	0.9626	0.8893	0.9727
	90	0.9566	0.8756	0.8606	0.8606
	80	0.9680	0.8446	0.9626	0.8893
Mylyn	100	0.2892	0.0027	0.0398	0.0355
	90	0.4046	0.0283	0.0158	0.0457
	80	0.0325	0.0227	0.0112	0.0459
ArgoUML	100	0.0047	0.0075	0.0081	0.0167
	90	0.0047	0.0315	0.0069	0.0181
	80	0.0103	0.0163	0.0085	0.0181
Freenet	100	0.0002	< 0.0001	< 0.0001	< 0.0001
	90	0.0074	< 0.0001	< 0.0001	< 0.0001
	80	0.0021	< 0.0001	< 0.0001	< 0.0001
JMeter	100	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	90	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	80	< 0.0001	< 0.0001	< 0.0001	0.0002

proposed approaches, E_{RL}-E_{MBA}, indicate that all of the lattice configurations considered yields statistically significant improvement in effectiveness while using concept lattices for concept location (see Table III).

5.5 Statistical significance of the effect of stemming

We investigate whether the improvements provided by stemming for the concept location technique using concept lattices is statistically significant or not. We observe the impact of stemming on the maximum possible precision gain (that is, the MPG). For this measure, we determine if stemming improves the results, that is, whether or not it increases MPG. Our goal is to determine if utilizing stemming *significantly* enhances the ability of generated concept lattices to reduce effort required during concept location.

In scenarios where stemming considerably improves the effectiveness of the baseline approach, concept lattices garner additional gains when the number of terms is large enough (that is, k>=15). We determine this by first, deriving the null hypothesis for each measure on all evaluated configurations for each software system. Since, for each configuration we have results for both non-stemmed and stemmed we use Wilcoxon's Matched-Pairs Signed-Ranks test with an alpha of 0.05. Table IV summarizes the results,

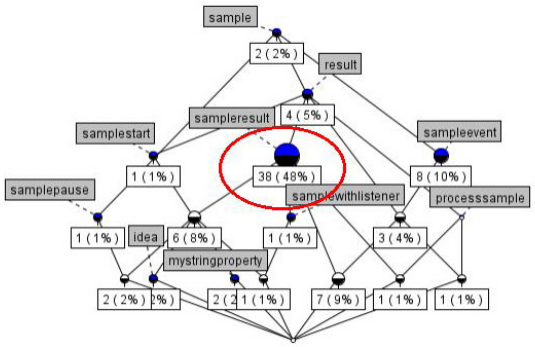

Fig. 11. Concept lattice for *JMeter* feature #195 using 80 documents and 10 terms. Each node represents a concept and the numbers associated with each node (located in the rectangles with white background) indicate number of methods assigned to the concept node. Rectangles with grey background are the terms associated with each concept node. The circled node shows the location of the first relevant method in the concept lattice.

where in the majority of cases the null hypothesis can be rejected for the significance level specified, indicating that our results are unlikely to be caused by chance. Results for all systems evaluated except iBatis support our intuition that stemming corpora significantly improves MPG. iBatis' p-values for maximum possible precision gain of various configurations are high, however. Further investigation reveals that for nine of the 13 features stemming improves the position of the first relevant method in the baseline technique. Additionally, without stemming, MBA values are relatively low, indicating excellent reduction in effort when searching for relevant information. The combination of these two factors considerably limits improvements possible by stemming while using concept lattices, which is confirmed by some of the p-values.

Overall, we conclude that stemming positively impacts both the concept location technique using concept lattices and the baseline approach. Analysis of the results (see Table IV) indicates that the reduction in effort (that is, methods viewed) while using concept lattices as compared to the baseline approach is statistically significant with an alpha level of 0.05 in the majority of the scenarios. For the software systems that were considered in our case study, the results show that p-values decrease as the number of terms increase. This appears to be caused by two factors: (1) stemming improves the position of the first relevant method for 56 out of 118 features and (2) the improvements obtained for configurations using stemming levels off quicker than results for non-stemmed as k increases. The first factor impacts the total amount of improvement possible. The combination of these two factors leads to the difference between the non-stemmed and stemmed results being less significant as the number of terms increases.

5.6 Outlier analysis

In this section we provide a few representative examples, which do not adhere to general patterns observed in the previous section. More specifically, we present the analysis of the results for two features from *Mylyn* and *JMeter*[17] (see the complete details for the concept lattices in Table V).

In case of the bug# 149838[18] for *Mylyn* we can observe that stemming negatively impacts the MPG values. For instance, for the lattice configuration of n=100 and k=25,

[17] http://www.cs.wm.edu/semeru/tosem-fca-lsi/ (*verified on 08/18/2011*)
[18] https://bugs.eclipse.org/bugs/show_bug.cgi?id=149838 (*verified on 08/18/2011*)

Table V. Experimental results for locating two features in *Mylyn* and *JMeter*. The table provides results for 12 non-stemmed and 12 stemmed configurations of concept lattices for each system.

			Non Stemmed				Stemmed			
Sys.	N	K	C	E_{MBA}	E_{RL}	MPG	C	E_{MBA}	E_{RL}	MPG
Mylyn (149838)	100	10	424	1	31	3000.0	232	1	6	500.0
	100	15	1202	1	31	3000.0	1331	1	6	500.0
	100	20	3285	1	31	3000.0	2914	1	6	500.0
	100	25	5142	1	31	3000.0	3477	1	6	500.0
	90	10	414	1	31	3000.0	228	1	6	500.0
	90	15	1055	1	31	3000.0	1119	1	6	500.0
	90	20	2085	1	31	3000.0	1674	1	6	500.0
	90	25	2629	1	31	3000.0	2007	1	6	500.0
	80	10	360	1	31	3000.0	185	1	6	500.0
	80	15	646	1	31	3000.0	734	1	6	500.0
	80	20	955	1	31	3000.0	1229	1	6	500.0
	80	25	1685	1	31	3000.0	1805	1	6	500.0
jMeter (195)	100	10	17	51	10	-80.4	31	28	10	-64.3
	100	15	28	49	10	-79.6	72	7	10	42.9
	100	20	43	37	10	-73.0	110	1	10	900.0
	100	25	64	32	10	-68.8	148	1	10	900.0
	90	10	17	47	10	-78.7	22	27	10	-63.0
	90	15	26	46	10	-78.3	60	7	10	42.9
	90	20	46	31	10	-67.7	85	1	10	900.0
	90	25	62	30	10	-66.7	108	1	10	900.0
	80	10	18	44	10	-77.3	31	10	10	0.0
	80	15	29	38	10	-73.7	46	1	10	900.0
	80	20	50	30	10	-66.7	80	1	10	900.0
	80	25	63	29	10	-65.5	112	1	10	900.0

the MPG values drop from 3000% to 500% after applying stemming. This case does not align with the majority of the cases from the previous sections, where stemming consistently improved the results of concept location using concept lattices. While this case negatively impacts the results, detailed analysis reveals that MPG values decrease since stemming already elevated the position of the first relevant method in the baseline techniques from the 31st to the 6th position (see the values of RL[19] in Table V). This case limits the potential of concept lattices to improve the results as the baseline technique already provides high rankings for relevant methods (the upper bound for MPG measure after stemming is 500%, whereas MPG values before stemming reach 3000%). In summary, this example emphasizes the importance of evaluating different measures in unison while assessing the effectiveness gain provided by stemming as in some cases stemming significantly improves the results of the baseline reducing the potential improvement acquired while using concept lattices.

Another case where we found MPG values to be negative is the situation where concept lattices contain relatively large concept nodes (see Fig. 11). Feature #195 (see online appendix) for *JMeter* is an example of such a situation. For this feature the lattice configuration of n=80 and k=10 terms, which appears in the Fig. 11, illustrates how large concept nodes affect the MPG values. Notably, the large concept nodes along the path of the minimal browsing area inflate the MBA value. The figure shows one relatively large

[19] RL is the position of the first relevant method in the ranked list of the results (see section 5.2.4 for complete definitions)

concept nodes, containing 38 methods (accounting for 48% of methods in lattice), which must be visited in order to reach the first relevant method. In this case, the node accounts for 38 of the 44 methods included in the minimal browsing area, unfavorably contributing to an MPG of -77.27%. For this particular scenario increasing the number of terms consistently reduces the number of methods of the MBA, illustrated by the MBA improving to 29 when 25 terms are used, but increasing terms does not present a universal or efficient solution to this problem. More research is required to better understand and address this type of cases, which are only a few in our data set. We have little reasons to believe that these cases are more prevalent in other systems; they are most likely also outliers. Applying more sophisticated term selection algorithms, such as the recently proposed approach of Kuhn [Kuhn 2009] which analyzes the log-likelihood of term frequencies, may alleviate the problem encountered in this example. Selection of appropriate terms may reduce the number of cases where several methods share the exact set of selected terms. We provide complete details on all the features and bugs (including other exceptional cases) in our online appendix.

5.7 Threats to validity

Several issues may have affected the results of the case study and thus may limit generalizations. We made all efforts to minimize the effect of these issues and we discuss them here.

One of the issues is that in our case studies we use the number of documents to build concept lattices that range from 80-100. However, if there are no relevant results in this range, we cannot compute any of the measures we used for evaluation. The assumption is that if the relevant results are ranked lower than 100, a new query needs to be formulated; nonetheless it is unlikely that a developer would inspect more than 100 methods before deciding to reformulate a query. Our evaluation measures take into account the number of nodes that need to be inspected. However, they do not take into account any costs associated while inspecting any individual nodes or elements in concept lattices or ranked lists of the results. In order to consider this information, user studies are required, which assume collecting low level information, such as interaction events, from within the IDE [Fritz et al. 2007]. We are planning on addressing this issue in our future work building on the results of prior user studies [de Alwis et al. 2007; Ko et al. 2006; Robillard et al. 2004; Starke et al. 2009]. A user study is also required to evaluate the practical usefulness of the attribute labels for navigational decisions during concept location. Currently, we assume that the insight provided by labels allows developers to make navigational decisions to reach the first relevant method with minimal effort, which introduces another threat to validity. In other words, our case study aims at evaluating a potential gain of using lattices as compared to the ranked lists of the results. The actual gain of using lattices can be evaluated via user studies where developers might use different strategies to browse and traverse lattices. Additionally, a user study would address the assumption we make that equal effort is required to evaluate a method, regardless of its size.

Although the software systems used in our evaluation come from a variety of application domains and differ in project size, they are all implemented in Java and are open source software systems. This issue prevents us from generalizing the results to software systems written in other programming languages and those systems developed as commercial software. On the other hand, there is no data to indicate how these factors would impact the results.

Another issue is the extent to which the software and the features used in the case study are representative of those actually used in practice. Although *ArgoUML, Freenet,*

iBatis, JMeter, Mylyn, and Rhino are real-world programs, this threat could be reduced if we experiment with other programs of different sizes and domains, as well as locating more concepts. In addition, the features related to the bugs used in our evaluation could have been implemented in more methods than those suggested in an official patch, as correcting the problem may involve only certain parts in the implementation. Once again, the assessment of the effectiveness gains remains valid, as both methods are equally influenced by this issue. The mappings for the features of *Rhino* and *iBatis* were derived by Eaddy et al. [Eaddy et al. 2008a; Eaddy et al. 2008b]. Additionally, the data provided for *iBatis* did not include descriptions of the concerns identified, which may have caused some imprecision in the results. For *Rhino*, the authors used the ECMA Specification to identify concepts and each concept was associated with a section in the documentation [Eaddy et al. 2008b]. There were no specifications available for *iBatis* so the authors derived concerns from the user's guide [Eaddy et al. 2008b]. We had to manually map concerns to sections of the user's guide, which described the concern. Although we tried to map concerns to sections where the name of the concern and the section heading from the user's guide indicate a relationship, it is possible that someone with more knowledge of the system would have been able to generate slightly different mappings. The resulting mappings could have negatively or positively impacted the results for *iBatis*.

5.8 Future work directions

The work done on this research thread so far provided solid results and also revealed directions where this research can move into. We plan to compare this approach with at least two other different strategies used to rank and select descriptive attributes to build concept lattices, for example, the terminological weighting formula and the Okapi [Cigarrán et al. 2004]. We also plan to design a heuristic-based approach to experiment with different strategies for splitting identifiers [Dit et al. 2011a; Enslen et al. 2009], mining abbreviations [Hill et al. 2008] or selecting attributes, which may be specific to source code, for example, selecting only attributes that represent data types or only class or methods names, etc. We plan to incorporate information about the rank of the method into the structure of the concept lattice, which may be helpful in terms of choosing the path while exploring a concept lattice. We will also investigate whether concept lattices can guide in re-ranking results of the IR-based concept location technique. With the emergence of various techniques to summarize software artifacts [Haiduc et al. 2010; Rastkar et al. 2010; Sridhara et al. 2010], we intend to explore the idea of replacing attribute labels with summaries of methods located in concept nodes of the lattice. Finally, we will investigate the impact of concept lattices on query reformulation strategies, which we did not address in this work.

6. CONCLUSIONS

In this paper we proposed a novel solution to address the problem of concept location in source code by combining Formal Concept Analysis and Information Retrieval. In the proposed approach, Latent Semantic Indexing is used to map concepts expressed in textual change requests (*e.g.*, bug reports or feature requests) to relevant parts of the source code, presented as a ranked list of search results. The benefit of our approach comes from automatically selecting most relevant attributes from a subset of source code documents in the search results and organizing them in a concept lattice using Formal Concept Analysis. We evaluated the proposed approach on six open-source systems with several hundred features and bugs for each system and derive our conclusions based on the analysis on different configurations of the corresponding concept lattices.

The proposed concept location method, which combines Information Retrieval and Formal Concept Analysis, provides very good results (*e.g.*, average *maximum possible precision gain* exceeding 2,000% for configurations of *Rhino*) when considering a relatively small subset of number of methods (*e.g.*, 100 out of 3,000 in the case of *Rhino*), hence it is easy to use for software of any size. It should be noted that our primary metric for comparing the proposed technique is based on *potential* optimistic gain while using concept lattices during concept location. Our approach outperforms in the best case scenario the IR-based concept location technique for all the configurations (*e.g.*, average *maximum possible precision gain* for all systems range between 71%-2,864% for configurations using 25 terms), whereas the best results are obtained while increasing the number of terms and decreasing the number of documents. Additionally, concept location with concept lattices using corpus stemming outperforms its carbon copy, which does not utilize stemming (as indicated by average improvement of 567% for all systems and configurations considered). Finally, concept lattices are shown to be quite effective (up to 79 times improvement over simple ranking) in terms of grouping relevant information and labeling topics, concepts, and relationships between them, offering the user additional cues when exploring the results of a search.

7. ACKNOWLEDGEMENTS

We are grateful to the anonymous TOSEM reviewers for their relevant and useful comments and suggestions, which helped us in significantly improving the earlier versions of this paper. This work is supported by NSF CCF-0916260, NSF CCF-1016868, NSF CCF-0845706, and NSF CCF-1017263 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

ANTONIOL, G. and GUÉHÉNEUC, Y.G. 2006. Feature Identification: An Epidemiological Metaphor. *IEEE Transactions on Software Engineering 32*, *9*, 627-641.

BACCHELLI, A., D'AMBROS, M. and LANZA, M. 2010a. Extracting Source Code from E-Mails. In *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC'10)*2010a.

BACCHELLI, A., D'AMBROS, M., LANZA, M. and ROBBES, R. 2009. Benchmarking Lightweight Techniques to Link E-Mails and Source Code. In *16th IEEE Working Conference on Reverse Engineering (WCRE'09)*, Lille, France, 205-214.

BACCHELLI, A., LANZA, M. and ROBBES, R. 2010b. Linking e-mails and source code artifacts. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, Cape Town, South Africa, 375-384.

BAEZA-YATES, R.A. and RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison-Wesley.

BIGGERSTAFF, T.J., MITBANDER, B.G. and WEBSTER, D.E. 1994. The Concept Assignment Problem in Program Understanding. In *15th IEEE/ACM International Conference on Software Engineering (ICSE'94)* 482-498.

CARPINETO, C., OSIŃSKI, S., ROMANO, G. and WEISS, D. 2009. A survey of Web clustering engines. *ACM Computing Surveys (CSUR) 41*, *3*.

CHEN, K. and RAJLICH, V. 2000. Case Study of Feature Location Using Dependence Graph. In *8th IEEE International Workshop on Program Comprehension (IWPC'00)*, Limerick, Ireland, 241-249.

CIGARRAN, J., PEÑAS, A., GONZALO, J. and VERDEJO, F. 2005. Evaluating Hierarchical Clustering of Search Results. In *12th International Conference on String Processing and Information Retrieval (SPIRE'05)*, 49-54.

CIGARRÁN, J.M., GONZALO, J., PEÑAS, A. and VERDEJO, F. 2004. Browsing Search Results via Formal Concept Analysis: Automatic Selection of Attributes. In *2nd International Conference on Formal Concept Analysis (ICFCA'04)*, Sydney, Australia, 74-87.

CLEARY, B., EXTON, C., BUCKLEY, J. and ENGLISH, M. 2009. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering 14*, *1*, 93-130.

CUBRANIC, D., MURPHY, G.C., SINGER, J. and BOOTH, K.S. 2005. Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering 31*, *6*, 446-465.

DE ALWIS, B., MURPHY, G.C. and ROBILLARD, M. 2007. A Comparative Study of Three Program Exploration Tools. In *15th IEEE International Conference on Program Comprehension*, 103-112.

DE LUCIA, A., FASANO, F., OLIVETO, R. and TORTORA, G. 2007. Recovering Traceability Links in Software Artifact Management Systems using Information Retrieval Methods. *ACM Transactions on Software Engineering and Methodology (TOSEM) 16*, *4*.

DE LUCIA, A., OLIVETO, R. and VORRARO, L. 2008. Using structural and semantic metrics to improve class cohesion. In *IEEE International Conference on Software Maintenance (ICSM'08)*, 27-36.

DEERWESTER, S., DUMAIS, S.T., FURNAS, G.W., LANDAUER, T.K. and HARSHMAN, R. 1990. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science 41*, *6*, 391-407.

DIT, B., GUERROUJ, L., POSHYVANYK, D. and ANTONIOL, G. 2011a. Can Better Identifier Splitting Techniques Help Feature Location? In *19th IEEE International Conference on Program Comprehension (ICPC'11)*, Kingston, Ontario, Canada, 11-20.

DIT, B., REVELLE, M., GETHERS, M. and POSHYVANYK, D. 2011b. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software Maintenance and Evolution: Research and Practice*.

EADDY, M., AHO, A.V., ANTONIOL, G. and GUÉHÉNEUC, Y.G. 2008a. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In *16th IEEE International Conference on Program Comprehension (ICPC'08)*, Amsterdam, The Netherlands, 53-62.

EADDY, M., ZIMMERMANN, T., SHERWOOD, K., GARG, V., MURPHY, G., NAGAPPAN, N. and AHO, A.V. 2008b. Do Crosscutting Concerns Cause Defects? *IEEE Transaction on Software Engineering 34*, *4*, 497-515.

EISENBARTH, T., KOSCHKE, R. and SIMON, D. 2003. Locating Features in Source Code. *IEEE Transactions on Software Engineering 29*, *3*, 210 - 224.

ENSLEN, E., HILL, E., POLLOCK, L. and VIJAY-SHANKER, K. 2009. Mining Source Code to Automatically Split Identifiers for Software Analysis. In *6th IEEE Working Conference on Mining Software Repositories (MSR'09)*, Vancouver, BC, Canada 71-80.

FERENC, R., SIKET, I. and GYIMOTHY, T. 2004. Extracting Facts from Open Source Software. In *20th IEEE International Conference on Software Maintenance (ICSM'04)* IEEE Computer Society: Los Alamitos CA, Chicago, Illinois, 60-69.

FLURI, B., WÜRSCH, M., GIGER, E. and GALL, H. 2009. Analyzing the Co-Evolution of Comments and Source Code. *Software Quality Journal 17*, *4*, 367-394.

FRITZ, T., MURPHY, G.C. and HILL, E. 2007. Does a programmer's activity indicate knowledge of code? In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 341 - 350.

GANTER, B. and WILLE, R. 1996. *Formal Concept Analysis*. Springer-Verlag, Berlin, Heidelberg, New York.

GAY, G., HAIDUC, S., MARCUS, M. and MENZIES, T. 2009. On the Use of Relevance Feedback in IR-Based Concept Location. In *25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Canada, 351-360.

GOLD, N., HARMAN, M., LI, Z. and MAHDAVI, K. 2006. Allowing Overlapping Boundaries in Source Code using a Search Based Approach to Concept Binding. In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, PA, 310-319.

GRANT, S., CORDY, J.R. and SKILLICORN, D.B. 2008. Automated Concept Location Using Independent Component Analysis In *15th Working Conference on Reverse Engineering (WCRE'08)*, Antwerp, Belgium, 138-142.

HAIDUC, S., APONTE, J., MORENO, L. and MARCUS, A. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *17th IEEE Working Conference on Reverse Engineering (WCRE'10)*, Beverly, Massachusetts, USA.

HAYES, J.H., DEKHTYAR, A. and SUNDARAM, S.K. 2006. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering 32*, *1*, 4-19.

HILL, E., FRY, Z.P., BOYD, H., SRIDHARA, G., NOVIKOVA, Y., POLLOCK, L. and VIJAY-SHANKER, K. 2008. AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools. In *5th Working Conference on Mining Software Repositories*, Leipzig, Germany.

HILL, E., POLLOCK, L. and VIJAY-SHANKER, K. 2007. Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, 14-23.

HILL, E., POLLOCK, L. and VIJAY-SHANKER, K. 2009. Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse. In *31st IEEE/ACM International Conference on Software Engineering (ICSE'09)*, Vancouver, British Columbia, Canada.

JIANG, H., NGUYEN, T., CHE, I.X., JAYGARL, H. and CHANG, C. 2008. Incremental Latent Semantic Indexing for Effective, Automatic Traceability Link Evolution Management. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, L'Aquila, Italy.

KERSTEN, M. and MURPHY, G.C. 2006. Using Task Context to Improve Programmer Productivity. In *14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Portland, Oregon, USA, 1-11.

KO, A.J., MYERS, B.A., COBLENZ, M.J. and AUNG, H.H. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering (TSE) 32*, *12*, 971-987.

KUHN, A. 2009. Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR'09)*2009.

KUHN, A., DUCASSE, S. and GÎRBA, T. 2007. Semantic Clustering: Identifying Topics in Source Code. *Information and Software Technology 49*, *3*, 230-243.

LIENHARD, A., DUCASSE, S. and AREVALO, G. 2005. Identifying Traits with Formal Concept Analysis. In *20th IEEE/ACM international Conference on Automated Software Engineering (ASE'05)*, Long Beach, CA, USA, 66 - 75.

LIU, D., MARCUS, A., POSHYVANYK, D. and RAJLICH, V. 2007. Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, Atlanta, Georgia, 234-243.

LO, K.K., CHAN, M.K. and BANIASSAD, E. 2006. Isolating and Relating Concerns in Requirements using Latent Semantic Analysis. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*, 383 - 396

LORMANS, M. and VAN DEURSEN, A. 2006. Can LSI help Reconstructing Requirements Traceability in Design and Test? In *10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, 47-56.

LUKINS, S., KRAFT, N. and ETZKORN, L. 2008. Source Code Retrieval for Bug Location Using Latent Dirichlet Allocation. In *15th Working Conference on Reverse Engineering (WCRE'08)*, Antwerp, Belgium, 155-164.

MALETIC, J.I., COLLARD, M.L. and MARCUS, A. 2002. Source Code Files as Structured Documents. In *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, Paris, France, 289-292.

MARCUS, A. and MALETIC, J.I. 2001. Identification of High-Level Concept Clones in Source Code. In *Automated Software Engineering (ASE'01)*, San Diego, CA, 107-114.

MARCUS, A., MALETIC, J.I. and SERGEYEV, A. 2005a. Recovery of Traceability Links Between Software Documentation and Source Code. *International Journal of Software Engineering and Knowledge Engineering 15*, *4*, 811-836.

MARCUS, A., POSHYVANYK, D. and FERENC, R. 2008. Using the Conceptual Cohesion of Classes for Fault Prediction in Object Oriented Systems. *IEEE Transactions on Software Engineering 34*, *2*, 287-300.

MARCUS, A., RAJLICH, V., BUCHTA, J., PETRENKO, M. and SERGEYEV, A. 2005b. Static Techniques for Concept Location in Object-Oriented Code. In *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, St. Louis, Missouri, USA, 33-42.

MARCUS, A., SERGEYEV, A., RAJLICH, V. and MALETIC, J. 2004. An Information Retrieval Approach to Concept Location in Source Code. In *11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, Delft, The Netherlands, 214-223.

MENS, K. and TOURWE, T. 2005. Delving source code with formal concept analysis. *Computer Languages, Systems & Structures 31*, *3-4*, 183-198.

POSHYVANYK, D., GUÉHÉNEUC, Y.G., MARCUS, A., ANTONIOL, G. and RAJLICH, V. 2007. Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering 33*, *6*, 420-432.

POSHYVANYK, D., MARCUS, A. and DONG, Y. 2006a. JIRiSS - an Eclipse plug-in for Source Code Exploration. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, 252-255.

POSHYVANYK, D., MARCUS, A., DONG, Y. and SERGEYEV, A. 2005. IRiSS - A Source Code Exploration Tool. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary, 69-72.

POSHYVANYK, D., MARCUS, A., FERENC, R. and GYIMÓTHY, T. 2009. Using Information Retrieval based Coupling Measures for Impact Analysis. *Empirical Software Engineering 14*, *1*, 5-32.

POSHYVANYK, D. and MARCUS, D. 2007. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, Banff, Alberta, Canada, 37-48.

POSHYVANYK, D., PETRENKO, M., MARCUS, A., XIE, X. and LIU, D. 2006b. Source Code Exploration with Google In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, PA, 334 - 338.

RAJLICH, V. and GOSAVI, P. 2004. Incremental Change in Object-Oriented Programming. In *IEEE Software*, 2-9.

RASTKAR, S., MURPHY, G. and MURRAY, G. 2010. Summarizing Software Artifacts: A Case Study of Bug Reports. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, 505-514.

RATIU, D. and DEISSENBOECK, F. 2007. From Reality to Programs and (Not Quite) Back Again. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, Banff, Alberta, Canada, 91-102.

REN, X., SHAH, F., TIP, F., RYDER, B.G. and CHESLEY, O. 2004. Chianti: a Tool for Change Impact Analysis of Java Programs. In *19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA '04)*, Vancouver, BC, Canada, 432-448.

REVELLE, M., DIT, B. and POSHYVANYK, D. 2010. Using Data Fusion and Web Mining to Support Feature Location in Software. In *18th IEEE International Conference on Program Comprehension (ICPC'10)*, Braga, Portugal, 14-23.

REVELLE, M. and POSHYVANYK, D. 2009. An Exploratory Study on Assessing Feature Location Techniques. In *17th IEEE International Conference on Program Comprehension (ICPC'09)*, Vancouver, British Columbia, Canada, 218-222.

ROBILLARD, M. 2005. Automatic Generation of Suggestions for Program Investigation. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, 11 - 20

ROBILLARD, M.P. 2008. Topology Analysis of Software Dependencies. *ACM Transactions on Software Engineering and Methodology 17*, *4*, 1-36.

ROBILLARD, M.P., COELHO, W. and MURPHY, G.C. 2004. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering (TSE) 30*, *12*, 889- 903.

SALTON, G. and MCGILL, M. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, NY, USA.

SAVAGE, T., REVELLE, M. and POSHYVANYK, D. 2010. FLAT^3: Feature Location and Textual Tracing Tool. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, Cape Town, South Africa, 255-258.

SHEPHERD, D. 2007. Natural Language Program Analysis: Combining Natural Language Processing with Program Analysis to Improve Software Maintenance Tools. In *Computer Science* University of Delaware, 176.

SHEPHERD, D., FRY, Z., GIBSON, E., POLLOCK, L. and VIJAY-SHANKER, K. 2007. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In *6th International Conference on Aspect Oriented Software Development (AOSD'07)*, 212-224.

SILLITO, J., MURPHY, G.C. and DE VOLDER, K. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering (TSE) 34*, *4*, 434-451.

SIMMONS, S., EDWARDS, D., WILDE, N., HOMAN, J. and GROBLE, M. 2006. Industrial tools for the feature location problem: an exploratory study. *Journal of Software Maintenance: Research and Practice 18*, *6*, 457-474.

SNELTING, G. 2005. Concept Lattices in Software Analysis. In *Formal Concept Analysis*, 272-287.

SRIDHARA, G., HILL, E., MUPPANENI, D., POLLOCK, L. and VIJAY-SHANKER, K. 2010. Towards Automatically Generating Comments for Java Methods. In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*.

STARKE, J., LUCE, C. and SILLITO, J. 2009. Searching and Skimming: An Exploratory Study. In *25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, Canada.

TAIRAS, R. and GRAY, J. 2009. An Information Retrieval Process to Aid in the Analysis of Code Clones. *Empirical Software Engineering 14*, *1*, 33-56.

TONELLA, P. 2003. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. *IEEE Transactions on Software Engineering 29*, *6*, 495-509.

TONELLA, P. and CECCATO, M. 2004. Aspect Mining through the Formal Concept Analysis of Execution Traces. In *11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, 112 - 121

VAN GEET, J. and DEMEYER, S. 2009. Feature Location in COBOL Mainframe Systems: an Experience Report In *25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, Canada, 361-370.

WILDE, N., BUCKELLEW, M., PAGE, H., RAJLICH, V. and POUNDS, L. 2003. A Comparison of Methods for Locating Features in Legacy Software. *Journal of Systems and Software 65*, *2*, 105-114.

WILDE, N., GOMEZ, J.A., GUST, T. and STRASBURG, D. 1992. Locating User Functionality in Old Code. In *IEEE International Conference on Software Maintenance (ICSM'92)*, Orlando, FL, 200-205.

YIN, R.K. 2003. *Applications of Case Study Research*. Sage Publications, Inc, CA, USA.

ZHAO, W., ZHANG, L., LIU, Y., SUN, J. and YANG, F. 2006. SNIAFL: Towards a Static Non-interactive Approach to Feature Location. *ACM Transactions on Software Engineering and Methodologies (TOSEM) 15*, *2*, 195-226.