Clustering Support for Static Concept Location in Source Code

Giuseppe Scanniello  
Dipartimento di Matematica e Informatica  
University of Basilicata  
Viale Dell'Ateneo, Macchia Romana  
giuseppe.scanniello@unibas.it

Andrian Marcus  
Department of Computer Science  
Wayne State University  
Detroit, USA  
amarcus@wayne.edu

Abstract— One of the most common comprehension activities undertaken by developers is concept location in source code. In the context of software change, concept location means finding locations in source code where changes are to be made in response to a modification request. Static techniques for concept location usually rely on searching the source code using textual information or on navigating the dependencies among software elements. In this paper we propose a novel static concept location technique, which leverages both the textual information present in the code and the structural dependencies between source code elements. The technique employs a textual search in that source code, which is clustered using the BorderFlow algorithm, based on combining both structural and textual data. We evaluated the technique against a text search based baseline approach using data on almost 200 changes from five software systems. The results indicate that the new approach outperforms the baseline and that improvements are still possible.

Keywords— Clustering, Lexical Analysis, Concept Location

I. INTRODUCTION

During software evolution, one of the most common developer activities is software change. Comprehension an essential part of software change. Developers need to understand as much of the software as necessary to make sure their changes are correct and do not affect the rest of the system adversely. Most changes start with a change request and once developers understand what is asked to do, they must find the places in the code where they have to implement the change. This task is viewed as two separate, yet related activities, that is, concept location and impact analysis [30]. During concept location developers start from the change request and end up locating a place in the source code where they will start the change. From this starting point, impact analysis is then used to determine what other places in the code are affected and how to carry out the complete change.

Concept location is a challenging activity as it aims to relate concepts expressed at different abstraction level and in different format. On one hand, developers have the change request (in the form of a new feature request or a bug description, etc.), usually formulated in natural language, and on the other hand, they have the source code, implemented in a programming language. Concept location is an instance of a more general problem coined concept assignment [4] and it is hence related to all instances of this problem, such as, traceability link recovery, for example. The main differences between all these instances lie in the context (i.e., change in this case) and the input and output generated.

Tool and process support for concept location in the context of software change has been the focus of much research. Section II discusses most approaches in more details. One way to distinguish between categories of concept location techniques is based on the type of information they use to help developers, that is, static or dynamic. Dynamic techniques rely on the execution of the system and the use of execution traces. Static techniques, on the other hand, rely on information extracted from the code without executing it. The two most common types of information used by static techniques are lexical (i.e., the text present in source code) and structural (i.e., dependencies present in source code). Most techniques focus on leveraging one such type of information, whereas a handfuls are concerned with their combination [10, 26, 32, 40].

In this paper, we present a new approach to static concept location that combines structural and lexical information. Our technique uses an Information Retrieval (IR) based approach [22], which formulates concept location is source code as a text retrieval problem. Simply put, the user writes a textual query based on their change request and retrieves source code documents (i.e., methods, classes, etc.). The approach’s simplicity is also its weakness. It relies solely on textual information and ignores structural information. Several extensions to this technique have been proposed to address this problem [10, 26, 29, 32, 40]. A common challenge to all these approaches is to determine the best way to combine these types of information. Our work takes another step in this research thread and the technique we propose is inspired from data mining approaches, where often times the data space is clustered before information is retrieved or searched for. In our context, the data space is the source code, which we cluster using the BorderFlow algorithm [23], based on combining both structural and textual data. Concept location is still formulated as a text retrieval problem, but instead of individual source code documents, the user retrieves clusters. We compare the new technique with the baseline IR-based approach using data associated with changes in response to 198 bug reports in the three software systems. The results indicate that the new technique outperforms the baseline in average. They also show that there is room for further improvement.
The remainder of the paper is organized as follows: related work is discussed in Section II, while the approach is presented in Section III. The case study and the discussion of the obtained results are presented in Section IV and Section V, respectively. Conclusions and future directions for our research sum up the paper in Section VI.

II. RELATED WORK

The work related to our approach falls in two categories, treated separately in this section: static concept location in source code and software clustering using lexical and structural information.

A. Concept Location

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 [30].

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. 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. [4] 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 build many useful add-ons on top of simple pattern matching, including references to class and method names, etc. Similarly, Raiti et al. [31] 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 [12, 17, 22, 28, 29] and natural language processing [15, 37] approaches, which allow more general textual queries and provide ranking of the results to these queries. Recent approaches also utilized independent component analysis [14], and Latent Dirichlet Allocation [18] 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. Our work tackles this limitation of existing IR-based concept location techniques, augmenting them with clustering using structural information in the code.

One of the first structural-based static techniques for concept location is the one proposed by Chen et al. [6], which is based on the search of an abstract system dependence graph. This approach has been recently extended [33] 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. [40] 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. [13] 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 concept location techniques can be found in [21].

B. Software Clustering

Approaches based on clustering algorithms have been widely employed in the software reverse engineering field [20, 38]. Specifically related to our work are the techniques that employ structural and lexical measures to support software clustering. For example, Anquetil and Lethbridge [1] present a comparative study of different hierarchical clustering algorithms and analyze their properties with regard to software remodularization. They also investigate the use of identifier names in this context. Lexical information is also leveraged in Kuhn et al. [16] where Latent Semantic Indexing (LSI) is used to generate similarity measures between documents for clustering. Corazza et al. [9] propose a lexical approach based on a customization of the well-known K-Medoids algorithm to partition Java software systems. This approach uses lexical information extracted from four zones in Java classes (i.e., comments, Javadoc, class/method names, and variable identifiers), which are weighted using a probabilistic model. The Expectation-Maximization (EM) algorithm is applied. Classes are then grouped using the clustering algorithm. The approach is assessed in a case study and the effect of using or not EM is investigated as well.

Earlier work by Maletic and Marcus [19] proposed an clustering approach based on the combination of lexical and structural information to identify abstract data types in legacy code. Structural and lexical information is also used to cluster methods of classes that are candidates for refactorings in the work of Bavota et al. [3] and to cluster classes of packages subject to remodularization [2].

More recently, Scanniello et al. [36] present a two phase approach for recovering hierarchical software architectures of object oriented software systems. The first phase uses structural information to identify software layers [35]. To this end, a customization of the Kleinberg algorithm is used. The second phase uses lexical information extracted from the source code to identify similarity among pairs of classes and then partitions each identified layer into software modules.
III. USING CLUSTERING AND TEXT RETRIEVAL FOR CONCEPT LOCATION

Our approach incorporates the IR-based concept location technique [22] and includes several additional steps. The concept location process, based on the new approach, consists of the following steps:

1. **Creating a corpus of a software system.** The source code is partitioned using a predetermined granularity level (i.e., 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. In this work we use method level granularity.

2. **Corpus normalization.** The resulting corpus can be normalized using a different set of techniques including: stop word removal, splitting identifiers, special token elimination, stemming, etc.

3. **Corpus indexing.** An IR engine is used to index the corpus. Different IR engines work in various ways, but most of them create and numerical index associated with each document in the corpus. Later, this index is used to determine similarity measures between documents. We used the Vector Space Model (VSM) [34] in this work.

4. **Computing semantic similarities between source code documents.** Depending on the IR engine used to index the corpus, a measure of lexical similarity between documents can be defined.

5. **Extracting dependencies in software.** Program dependencies are extracted statically from the source code, at the same granularity level as the source code corpus (i.e., class or method). Depending on the availability of analysis tools, more or less complex structural relationships can be extracted, such as: direct method calls, common attribute reference, inheritance, etc. In this work, we used static method references.

6. **Clustering.** The software system is clustered using the lexical similarities between source code documents and the structural dependencies between them. In this work we chose the BorderFlow clustering algorithm [24].

7. **Formulating a query.** The approach supports textual queries. Developers formulate such queries based on the information they have about the change request. Most IR engines do not rely on a predefined vocabulary or grammar; hence the queries do not need to be correct sentences. The query is usually normalized the same way the corpus has been normalized.

8. **Ranking documents.** The queries are projected into the document space generated by the IR engine. Then lexical similarities between the user query and the documents from the source code (methods in our case) are computed. Similarities between the query and each cluster are also computed. Clusters are retrieved based on their similarity to the query. Inside each cluster, the documents (methods) are ranked by their similarity to the query in descending order.

9. **Examining results.** The developer investigates the results in the order they are retrieved. If a user finds a part of the concept (that is, the location of the change to be done), then the search succeeds, otherwise, the user formulates a new query, taking into account new knowledge obtained from the investigated documents, and returns to step 7.

Steps 1, 2, 3, 7, 8, and 9 are part of the baseline IR-based approach (although step 8 is changed significantly here), whereas the other steps (4, 5, and 6) are new to our approach. The remainder of the section describes in details how we instantiated each step in this work. Investigating alternative instances is subject of future work.

**A. Corpus Creation**

We use a method level granularity in this application, as it is common in IR-based concept location applications. Each method results in one document in the corpus. All the comments and identifiers from the implementation of the method are included in the document. Lead comments for the methods (if any) were also included in the corresponding document.

**B. Corpus Normalization**

The normalization is performed deleting non-textual tokens (i.e., operators, special symbols, numbers, etc.), splitting terms composed of two or more words (e.g., “first name” is transformed into “first” and “name”) and eliminating all the terms within a stop word list and with a length less than three characters. The stop word list includes also the keywords of the Java programming language. Finally, the Porter stemmer [27] is applied on the lexical items to reduce words to their root forms. For instance, both the words designing and designer lead to the common radix design.

**C. Corpus Indexing**

In this work we use one of the most popular IR technique used in text retrieval, the Vector Space Model (VSM) [34]. VSM is a statistical corpus based method, based on the occurrences of words in documents from the corpus. It does not use a predefined vocabulary or grammar, so it can be easily applied to any kind of corpora.

VSM relies on the construction of a term-by-document matrix, where a generic entry denotes the occurrence of a term in a given document. The document model assumed by this representation is called bag-of-words, since each document is represented as a multi-set of words, disregarding all information about their order or syntactic structure. For the weight associated to each pair (term, document) we consider term frequency–inverse document frequency, also known as *tf-idf*, which is defined as follows for every term $t$ and document $d$:

$$
tf - idf(t_i, d_j) = tf(t_i, d_j) \cdot \frac{\log \frac{|C|}{df(t_i)}}
$$

where $tf(t_i, d_j)$ is the number of occurrences of the term $i$ in the document $j$, while $|C|$ is the total number of documents.
and $df(t_i)$ is the number of documents containing the term $t_i$. Note that $tf-idf$ is equal to zero when $t_i$ does not appear in the document or it appears in all the documents (the presence of the term in the document is irrelevant). Higher values are assigned either to terms with a high number of occurrences in the document or to terms that appear in a small number of documents. In this model, each document is associated with a real valued vector that spans the space of terms in the corpus (i.e., a row in the term-by-document matrix).

D. Computing Lexical Similarities Between Source Code Documents

This step is not used in the baseline IR-based approach. It is a necessary step in order to perform the clustering. Several similarity measures between documents can be defined using the VSM representation. One of the most common similarity measures used in text retrieval applications of VSM is the cosine between the vectors corresponding to two documents. We refer to this measure as lexical or semantic similarity between documents and compute it between all the documents in the corpus (i.e., methods in the software).

E. Extracting Dependencies in Software

We represent the software system as a directed graph $G = (V, E)$. $V$ represents the methods in the system, while $E$ is the set of edges (i.e., ordered pair of elements of $V$). Note that each edge $(m_i, m_j) \in E$ represents a directed relationship between two methods, $m_i$ and $m_j$, if there is a dependency relationship between them. We take a conservative approach in this work and only consider direct references between methods. In other words, $(m_i, m_j) \in E$, if there is a reference to $m_i$ in the body of $m_j$. We use JRipples [5] to identify these dependencies.

Clearly this is a subset of the structural relationships between methods in OO software. Including additional relationships in the program dependence graph is subject of future work.

F. Clustering

The directed graph $G = (V, E)$ is enhanced using the similarity lexical measures computed between each method pair. In particular, this graph is turned into the directed weighted graph $G' = (V, E, \omega)$ by adding a weight to each edge $(m_i, m_j) \in E$. The weight of the edge $(m_i, m_j)$ is the semantic similarity between $m_i$ and $m_j$, computed as the cosine between their corresponding vectors in the VSM representation. The graph $G'$ summarizes both the structural and the lexical information of a software system.

The BorderFlow clustering algorithm [23] is applied to $G'$. The algorithm is a general-purpose graph clustering algorithm originally conceived for achieving a soft clustering of the input graph (i.e., a node can be in one or more clusters), but it can also be used for hard clustering (i.e., a node can be in exactly one cluster) [23]. This algorithm has been selected since it has been successfully applied in the past to natural language processing problems and to extract concepts from large word similarity graphs [24]. We also choose it because the graph we generate is sparse and most other existing graph based clustering algorithms work best on dense graphs.

The formal definition of the clustering algorithm is provided in [11], where a cluster is defined as collection of nodes, such that each cluster has more links inside than links to the outside. Further on, a cluster has a set of nodes such that the flow is maximal in the cluster, while the flow from the cluster to the other ones is minimal. The idea behind border flow clustering algorithm is to maximize the flow from the border of each cluster to its inner nodes (i.e., the nodes within the cluster) while minimizing the flow from the cluster to the nodes outside of the cluster.

The border flow clustering can be applied on any weighted graph $G = (V, E, \omega)$. The weighing function $\omega$ assigns a positive weight to each edge of $E$. In case an edge between two nodes does not exist the algorithm assumes that non-existing edges are edges $e$ such that $\omega(e) = 0$. A cluster $X$ is a subset of $V$ such that it maximizes the border flow ratio:

$$F(X) = \frac{\Omega(b(X), X)}{\Omega(b(X), n(X))}$$

where $b(X)$ is the set of border nodes of $X$, while $n(X)$ is the set of direct neighbors of $X$. On the other hand, $\Omega$ is a function that assigns the total weight of the edges from a subset of $V$ to another one to these subsets (i.e., the flow between the first and the second subset):

$$\Omega : 2^V \times 2^V \rightarrow R, \quad \Omega(X, Y) = \sum_{x \in X, y \in Y} \omega(x, y)$$

The idea behind the algorithm is to iteratively select elements from $n(X)$ and then to insert them in $X$ until $F(X)$ is maximized. The selection of the nodes to be inserted at each iteration is carried out according to the following two steps:

1. Computing the set $C(X)$. It contains all the nodes $u \in X \cup \{u\}$ such that $F(X \cup \{u\}) > F(X)$

2. Select the candidates $u \in C(X)$ to get the set $C_f(X)$. This set contains all the nodes $u$ that maximize $\Omega(u, n(X))$

All elements of $C_f(X)$ are then inserted in $X$ if the following condition is verified $F(X \cup C_f(X)) \geq F(X)$.

In order to reduce the computation time some heuristics have been proposed. The worst-case time complexity of the BorderFlow algorithm is $O(n^3)$, where $n$ is the number of nodes. In average, it is linear in the number of edges. The heuristic version has the same theoretical characteristics but a very different empirical behavior. The interested reader can found further details on both the general definition of the algorithm and the proposed heuristics in [23].

The rationale for employing this algorithm relies on the fact that it is unsupervised, namely it does not require fixing the number of clusters to be computed. A possible drawback of this algorithm is that it could identify partitions on the same graph that present some slight differences. This happens when the algorithm is applied to get hard clustering of large graphs. A possible explanation is that the used implementation is based on heuristics to reduce the
computation time. The implementation used in this work is available at http://borderflow.sourceforge.net/.

G. Document Ranking

In the IR-based baseline approach documents are retrieved based on their lexical similarity to the query. In our approach, the position of the methods in the ranked list of results is modified according to the clustering results. Specifically, for all the clusters \( c_k \in C \), we compute the similarity of each method \( m_i \in c_k \) with the query \( q \) as follows:

\[
S(q, m_i, c_k) = \max_{m_j \in C_k} \cos(q, m_j)
\]

where \( \cos(q, m_j) \) is the lexical similarity between then query \( q \) and method \( m_j \).

The methods are then sorted to get a new ranked list. Note that the methods maintain their relative order within the cluster. In essence, the new approach no longer retrieves individual methods but instead clusters of related methods (related both structurally and lexically). The retrieval order of the clusters is still based on the lexical similarity to the user query.

IV. EVALUATION

We implemented a tool based on the above instantiation of our technique and conducted an empirical study to evaluate the new approach. This section presents the design underlying the empirical investigation following the guideline provided by Wohlin et al. in [39].

A. Definition and Context

The goal of the study was to analyze whether the accuracy of locating features in source code improves using the proposed approach compared to the IR-based baseline approach. Accordingly, we have formulated and investigated the following research question:

Does the new approach outperform the baseline approach?

The perspective of the study is both from the point of view of the researcher, evaluating whether the concept location improves using the new approach, and from the point of view of the project manager, who wants to evaluate the possibility of adopting the proposed approach within his/her own organization.

The study has been conducted on five open source Java software systems (although we used two versions of one of the systems). These systems have change data available for 198 of bug reports, published in [25]. For each bug report a set of methods that were changed in order to fix the problem is identified.

Information on these systems and the number of bugs considered in the case is shown in Table I. The first column shows the name of the system and the URL of the official web page. The analyzed versions of each system and the number of classes are reported in the second and third columns, respectively. The number of lines of codes (KLOCs) is shown in the fourth column, while the number of methods is presented in the fifth column. The number of bug reports used in the study is reported in the sixth column. Finally, a short summary of the functionality of the system is in the last columns.

B. Planning

Our choice of empirical evaluation is based on reenacting concept location based on past changes and to simulate the user 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: (1) the query formulation and reformulation (when needed); and (2) 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 bug description. These reports contain both the title

<table>
<thead>
<tr>
<th>System</th>
<th>Vers.</th>
<th>#Class</th>
<th>KLOCs</th>
<th>#Methods</th>
<th># Bugs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Art of Illusion</td>
<td>2.4.1</td>
<td>453</td>
<td>103.4</td>
<td>5254</td>
<td>12</td>
<td>A free, open source 3D modeling and rendering studio.</td>
</tr>
<tr>
<td>ATunes</td>
<td>1.10</td>
<td>419</td>
<td>42.3</td>
<td>3406</td>
<td>30</td>
<td>A full featured audio player and manager</td>
</tr>
<tr>
<td>Eclipse</td>
<td>3.5</td>
<td>18956</td>
<td>2599.5</td>
<td>169222</td>
<td>114</td>
<td>An open development platform for building, deploying and managing software across the lifecycle</td>
</tr>
<tr>
<td>JEdit</td>
<td>4.2</td>
<td>411</td>
<td>94.5</td>
<td>3,696</td>
<td>33</td>
<td>A text editor for programming with an extensible plug-in architecture.</td>
</tr>
<tr>
<td></td>
<td>4.3</td>
<td>492</td>
<td>101.3</td>
<td>4588</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
of the bugs and their description, which are referred to change requests of faulty features.

In this study we created queries considering three different options: (i) title of the bug; (ii) description of the bug; and (iii) concatenation of title and description. For example, the title and a description of a bug for Eclipse 3.5 are: "Ant launch configs should run in separate JRE by default" and "Using 3.5.x, when I run an Ant build file from the launch shortcuts, it is run in a separate JRE by default. Using 3.6, it is run in the same JRE as Eclipse". The concatenation of the title and the description is performed attaching the description to the title. Note that the way how these title and description are concatenate does not affect the feature location results. This is because the query vector is built disregarding all information about the query words and their possible syntactic structure.

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 approach. 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, otherwise not possible.

Based on our choices and assumptions, the empirical evaluation consists of a case study where our new concept location 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 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).

As mentioned before, the baseline approach is the IR-based concept location without the clustering. The technique is instantiated here the same way we did it with the new approach (i.e., using the same corpus creation process, corpus normalization, and VSM as the IR engine). For simplicity and readability, we will refer to the baseline approach as simply baseline and to the new approach as CLC (Concept Location using Clustering).

C. Hypotheses Formulation

We formulated and tested the three null hypotheses, which are one-sided since we expected a positive effect of combining the lexical and structural information:

- **H_{a1}.** CLC does not significantly improve the results compared to the baseline when using the title of a bug as the query.
- **H_{a2}.** CLC does not significantly improve the results compared to the baseline when using together the bug title and its description as the query.
- **H_{a3}.** CLC significantly improves the results compared to the baseline when using the title of a bug as the query.
- **H_{a4}.** CLC significantly improves the results compared to the baseline when using the description of a bug as the query.
- **H_{a5}.** CLC significantly improves the results compared to the baseline when using together the bug title and its description as the query.

In case the null hypothesis can be rejected with a relatively high confidence it is possible to accept an alternative hypothesis, which admits a positive effect of CLC as compared to the baseline. The related alternative hypotheses are:

- **H_{a1}.** CLC significantly improves the results compared to the baseline when using the title of a bug as the query.
- **H_{a2}.** CLC significantly improves the results compared to the baseline when using the description of a bug as the query.
- **H_{a3}.** CLC significantly improves the results compared to the baseline when using together the bug title and its description as the query.

D. Selected Variables

The main factor on which our study is focused on is Method. It is a nominal variable that assumes as values CLC and baseline. In order to better assess the effect of Method we also consider the way how the queries are formulated and used. To this end we considered the factor Query that is a nominal variable that assumes the following values: Title, Descr, and Title+Descr. We have also analyzed the effect of the systems used in the study on the experimental results. To this end, the factor System has been controlled. It can assume one of the following values: ArtOfIllusion, ATunes, Eclipse, JEdit4.2, and JEdit4.3. We used two versions of JEdit and we considered them as different systems.

It is common in IR tasks to use recall and precision as performance measures. IR-based concept location is an instance of an IR concerned with the retrieval of a single document (i.e., where the change starts). In this context, once the target document is identified, recall is 1 (i.e., highest value). At recall value of 1, precision becomes the relevant measure and it is common to use its inverse in concept location studies, referred to as effectiveness [28]. It indicates the position of the first relevant method from the gold set. This measure is used as an approximation of the effort it takes to locate a concept, assuming each document takes a unit of effort to investigate. When comparing two concept location techniques using reenactment, each changed method is ranked based on the query and the highest rank among the changed methods is the effectiveness (or effort requirement) of the technique. Since this is an inverse measure (i.e., the inverse of the precision), the lower the effectiveness (the effort) the better the technique is.

In this study, since the BorderFlow clustering algorithm may produce different results, we executed each query five times and then collected the positions of the first relevant method from the changed set for each run. The median of these values has been used to compare the results with respect to the ones achieved with baseline.

The position of the first changed method (for both CLC and baseline) is the dependent variable used in the study and is called effectiveness in the rest of the paper.
E. Execution and Data Analysis

We applied CLC and the baseline on each system using all the different options in issuing the queries and then measured the effectiveness of each run.

We adopted non parametric tests to reject the null hypotheses. In particular we selected Mann-Whitney test [8] for unpaired analysis because it is very robust. We decided (as it is customary) to accept a probability of 5% of committing Type-I-error [39], i.e., rejecting the null hypothesis when the alternative one is actually true.

While the statistical tests allow for checking the presence of significant differences, they do not provide any information about the magnitude of such a difference. In other words, the analysis of the effect sizes facilitates the interpretation of the substantive as opposed to the significance difference of the results of a statistical analysis. We used here the Cohen’s \( d \) standardized difference between two groups [7]. In the context of unpaired analyses, we computed (as usual) Cohen’s \( d \) as the difference between the means of the distributions divided by the pooled standard deviation. The difference is considered negligible for \(|d| < 0.2\), small for \(0.2 \leq |d| < 0.5\), medium for \(0.5 \leq |d| < 0.8\), and large for \(|d| \geq 0.8\).

V. Results

Some descriptive statistics (i.e., min, max, mean, median, and standard deviation) of the results (i.e., effectiveness) achieved by applying CLC and baseline on the selected software systems are summarized in Table III and Table IV, respectively. Descriptive statistics of all the software systems together are shown as well. The cases where the mean and median are better for CLC than the baseline are highlighted in Table III.

A quick analysis of the results in the two tables allows us to note that in average CLC outperforms the baseline and that in most cases using the title only as the query leads to better results (in average) than the other two options. A more detailed analysis of the results is presented in the remainder of this section.

<table>
<thead>
<tr>
<th>System</th>
<th>Query</th>
<th>Hypotheses</th>
<th>p-value</th>
<th>Cohen’s “d”</th>
<th># of CLC &lt; baseline</th>
<th># of CLC &gt; baseline</th>
<th># of CLC = baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>Title</td>
<td>H₀₀</td>
<td>0.04 (YES)</td>
<td>-0.04</td>
<td>109</td>
<td>88</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>H₀₁</td>
<td>&lt;0.01 (YES)</td>
<td>-0.11</td>
<td>122</td>
<td>74</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>H₀₂</td>
<td>&lt;0.01 (YES)</td>
<td>-0.09</td>
<td>113</td>
<td>83</td>
<td>2</td>
</tr>
<tr>
<td>Art of Illusion</td>
<td>Title</td>
<td>H₀₀</td>
<td>0.10 (NO)</td>
<td>-0.33</td>
<td>9</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>H₀₁</td>
<td>0.27 (NO)</td>
<td>-0.27</td>
<td>10</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>H₀₂</td>
<td>0.13 (NO)</td>
<td>-0.57</td>
<td>10</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ATunes</td>
<td>Title</td>
<td>H₀₀</td>
<td>0.37 (NO)</td>
<td>-0.11</td>
<td>15</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>H₀₁</td>
<td>0.34 (NO)</td>
<td>0.05</td>
<td>17</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>H₀₂</td>
<td>0.46 (NO)</td>
<td>0.08</td>
<td>14</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Title</td>
<td>H₀₀</td>
<td>0.12 (NO)</td>
<td>-0.04</td>
<td>57</td>
<td>56</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>H₀₁</td>
<td>0.04 (YES)</td>
<td>-0.22</td>
<td>67</td>
<td>45</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>H₀₂</td>
<td>0.10 (NO)</td>
<td>-0.15</td>
<td>54</td>
<td>58</td>
<td>2</td>
</tr>
<tr>
<td>JEdit 4.2</td>
<td>Title</td>
<td>H₀₀</td>
<td>0.18 (NO)</td>
<td>-0.20</td>
<td>23</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>H₀₁</td>
<td>0.22 (NO)</td>
<td>0.05</td>
<td>19</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>H₀₂</td>
<td>0.14 (NO)</td>
<td>0.03</td>
<td>19</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>JEdit 4.3</td>
<td>Title</td>
<td>H₀₀</td>
<td>0.26 (NO)</td>
<td>-0.11</td>
<td>5</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>H₀₁</td>
<td>0.02 (YES)</td>
<td>-0.96</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>H₀₂</td>
<td>0.04 (YES)</td>
<td>-0.68</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table II. Data Analysis results

Table III. Descriptive Statistics for CLC

<table>
<thead>
<tr>
<th>System</th>
<th>Query</th>
<th>Min</th>
<th>Max</th>
<th>Med</th>
<th>Mean</th>
<th>StdD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>Title</td>
<td>4273</td>
<td>177</td>
<td>524</td>
<td>823.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>4056</td>
<td>184</td>
<td>638.4</td>
<td>914.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>4056</td>
<td>161</td>
<td>577.9</td>
<td>888.1</td>
<td></td>
</tr>
<tr>
<td>Art of Illusion</td>
<td>Title</td>
<td>3043</td>
<td>278</td>
<td>991.2</td>
<td>1262.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>3421</td>
<td>1504</td>
<td>1509</td>
<td>1164.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>3424</td>
<td>1078</td>
<td>1278</td>
<td>1281.5</td>
<td></td>
</tr>
<tr>
<td>ATunes</td>
<td>Title</td>
<td>2742</td>
<td>251</td>
<td>758.8</td>
<td>921.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>3648</td>
<td>558</td>
<td>1038</td>
<td>1222.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>3617</td>
<td>461</td>
<td>926.6</td>
<td>1147.0</td>
<td></td>
</tr>
<tr>
<td>Eclipse</td>
<td>Title</td>
<td>1785</td>
<td>138</td>
<td>289.6</td>
<td>406.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>1542</td>
<td>115</td>
<td>286.7</td>
<td>381.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>1770</td>
<td>117</td>
<td>273.4</td>
<td>378.5</td>
<td></td>
</tr>
<tr>
<td>JEdit 4.2</td>
<td>Title</td>
<td>4273</td>
<td>330</td>
<td>1030</td>
<td>1273.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>4056</td>
<td>654</td>
<td>1050</td>
<td>184.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>4056</td>
<td>459</td>
<td>983.2</td>
<td>1212.9</td>
<td></td>
</tr>
<tr>
<td>JEdit 4.3</td>
<td>Title</td>
<td>18</td>
<td>238.9</td>
<td>378.5</td>
<td>376.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Descr</td>
<td>19</td>
<td>1094</td>
<td>1109.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Title+Descr</td>
<td>2610</td>
<td>369</td>
<td>851.4</td>
<td>1101.7</td>
<td></td>
</tr>
</tbody>
</table>

Table IV. Descriptive Statistics for Baseline
A. Research Question: CLC vs. Baseline

The statistical data analysis is summarized in Table IV. In particular, the first column shows the name of the software system, while the Query values, while the corresponding null hypothesis is indicated in the second column. The p-values (whether the null hypotheses can be rejected is shown as well) and the values of the Cohen’s $d$ effect size are reported in the third and fourth columns, respectively. The fourth column shows the number of times that CLC produced better results ($\#$ of CLC < baseline) than the baseline, while the fifth column reports the number of times that better results have been achieved using baseline ($\#$ of CLC > baseline). The number of time that CLC and baseline produced the same results ($\#$ of CLC = baseline) is indicated in the last column.

The results of the Mann-Whitney test reveal that the null hypotheses $H_{0t}$, $H_{1t}$, and $H_{2t}$ can be rejected when the data of all the systems are considered together. Therefore, we can state that CLC significantly improves the results compared to the baseline, when the title and the description of a bug are used alone or together as the query. As far as the effect size on Title is concerned, we obtained a standardized Cohen’s $d = -0.04$, which can be considered negligible in practical use. The effect size was negligible also for Descr and Title+Descr (-0.11 and -0.09, respectively).

CLC produced significantly better results also on Eclipse when using the description of a bug as query (p-value = 0.04). The standardized effect size is small (Cohen’s $d = -0.22$). Regarding JEdit4.3, a statistically significantly different can be observed when using CLC and baseline on Descr and Title+Descr. The effect size was large on Descr (Cohen’s $d = -0.96$), while was medium on Title+Descr (Cohen’s $d = -0.68$).

The number of times that CLC produced better results was larger than the number of times that better results have been achieved using baseline. This is true both considering the software systems alone and together. However, on both Eclipse and ATunes using Title+Descr the number of times that baseline produced better results with respect to CLC is slightly larger. Although the analysis of the effect size is mainly used to complement inferential statistics in case a significant difference is present (in our case p-value is larger than 0.05), the data analysis shows that the effect size is also small (Cohen’s $d = -0.15$) in favor of CLC on Eclipse using Title+Descr. On the other hand, the effect size is negligible on ATunes, thus indicating equivalence between the two methods, in this case.

We also looked at the magnitude of the differences between performance (i.e., effectiveness) of the two techniques. Table V summarizes the average and median differences between the two techniques in the cases when one or the other performs better.

Our approach is geared to alleviate a specific problem exhibited by the baseline approach, that is, the situation when the top ranked methods are the result of noise in the system and are not in fact related to the change set. The intuition behind the new approach is that in such cases, these methods will cluster with other non relevant methods and their cluster will rank lower than other clusters that contain relevant methods. With this in mind, the way the clusters are constructed and the way the similarity between a query and a cluster is constructed may have a large influence on the result. Our choice of maximum similarity to compute the similarity between a query and a cluster leads arguably to the smallest change of the original list at the very top and hence the main influence we observe is that of the clustering technique. It is interesting to note that the BorderFlow algorithm produced in average rather small clusters. Specifically, the size of the clusters across all systems ranged from 1 to 84 methods, with an average size of 3 methods. We expect that larger cluster sizes in average would result better results. It may be undesirable to retrieve clusters that are too large, as the user would have a hard time analyzing them. Determining the optimal cluster size is subject of future work.

Based on the current results, we can positively answer our research question and conclude that clustering the system based on structural and lexical data prior to using IR-based concept location leads to better results than the alternative
An approach based on lexical and structural information is performed manually. In this context, a novel and automate software system, feature location can be a laborious task if each case. The current results indicate that this aspect influence greatly the results. Studies with developers will also allow us to collect qualitative data with respect to the quality of the clusters produced. We plan to conduct such studies in the future.

The choice of the IR engine to use may also affect the results. It is conceivable that different IR engines may be more or less sensitive to the clustering results. Another issue is represented by the fact that the used clustering algorithm identifies partitions that present some slight differences on the same software system between subsequent runs. To reduce biases, we executed each query five times. Then we collected the positions of the first relevant method from the change set for each run and the median of these values has been used in the study. Further details on the results of each run are not presented for space reasons. Our choice of five runs is rather arbitrary and more runs could lead to different results.

The software systems we used in our empirical study may affect the generalization of the results. For example, the use of open source software systems could threaten the validity of the results. In fact, in contrast with more centralized models of development such as those typically used in commercial software companies, this kind of systems are mainly developed according to a mass collaboration. Accordingly, a further investigation on commercial software systems is needed to increase our awareness on the achieved results. This phase of our research plan is still in progress and represents the most challenging. Another issue regards the size of the used software systems. The effect of larger software systems should be also investigated, as well as the effect of the programming language and paradigm used. All the systems we used are written in Java.

Further threats concern the validity of the statistical tests. Statistical non-parametric tests (Mann-Whitney and Wilcoxon) were used to reject the null hypotheses. This test is quite robust and has been extensively used in the past to conduct analysis similar to ours.

Finally, we did not perform a comparison of our proposal with other approaches for concept locations on a public dataset. This was due to the lack of such a dataset.

VI. CONCLUSIONS AND FUTURE WORK

For software maintainers who are unfamiliar with a software system, feature location can be a laborious task if performed manually. In this context, a novel and automate approach based on lexical and structural information is proposed in this paper to support the location of concepts in existing source code. In particular, the approach combines lexical and structural information and it uses the BorderFlow clustering algorithm to group methods. Successively, all the identified groups are used to identify source code that is similar to a query describing the feature to be searched.

To assess the approach and the underlying techniques, we implemented a prototype of a supporting software system. Even if the approach is general and has been conceived to analyze software systems implemented using any object oriented programming language, we have implemented and then used a prototype for Java software systems. The approach has been validated in a case study using changes in response to 198 bug reports in five open source software systems implemented in Java. The data analysis has revealed that the approach produces better results than the baseline approach. In particular, a significant statistical difference was observed in favor of our approach. The study also revealed that the way in which the query is formulated has an impact on the results.

Several aspects of our new approach deserve further work, which we plan to undertake. We plan to implement variations of the approach using different clustering techniques, as well as different distance measures. We also plan to use and compare different IR engines in this context. Qualitative data will be collected through studies involving actual users, especially with regard to the generated clusters.

The current results give no indication on the performance of the technique of different software systems. We plan to focus future analysis on using systems with various different properties. This part of our research is actually the most demanding and challenging. Extracting enough data for reenactment of concept location is not easy and few sources of such data exist. In reviewing related work, we found that our study is one of the largest in terms of number of data points (changes) collected.

ACKNOWLEDGEMENTS

We would like to thank the Michele Brescia and Daniele Pascale, who developed some of the software modules of the prototype implementing the approach presented here. Special thanks are due to Alex-Cyrille Ngonga Ngomo for his support with the BorderFlow clustering algorithm.

Andrian Marcus was supported in part by grants from the US NSF: CCF-1017263, CCF-0845706, and CCF-0820133.

REFERENCES
