On the Conceptual Cohesion of Co-Change Clusters

Marcos César de Oliveira*†, Rodrigo Bonifácio†, Guilherme N. Ramos† and Márcio Ribeiro‡
*Federal Budget Secretary
Ministry of Planning, Budget and Management, Brazil
Email: marcos-cesar.oliveira@planejamento.gov.br
†Department of Computer Science, University of Brasília
E-mail: {rbonifacio,gnramos}@cic.unb.br
‡Computing Institute, Federal University of Alagoas
E-mail: marcio@ic.ufal.br

***Abstract*—The analysis of co-change clusters as an alternative software decomposition can provide insights on different perspectives of modularity. But the usual approach using coarse-grained entities does not provide relevant information, like the conceptual cohesion of the modular abstractions that emerge from co-change clusters. This work presents a novel approach to analyze the conceptual cohesion of the source-code associated with co-change clusters of fine-grained entities. We obtain from the change history information found in version control systems. We describe the use of our approach to analyze six well established and currently active open-source projects from different domains and one of the most relevant systems of the Brazilian Government for the financial domain. The results show that co-change clusters offer a new perspective on the code based on groups with high conceptual cohesion between its entities (up to 69% more than the original package decomposition), and, thus, are suited to detect concepts pervaded on codebases, opening new possibilities of comprehension of source-code by means of the concepts embodied in the co-change clusters.**

I. Introduction

Several approaches for software comprehension have been proposed to help developers to understand the decomposition of a system under different perspectives—instead of limiting the analysis to the typical representation based on the structure and usage relations of the software components. Actually, this static representation introduces several limitations. First, the design structure of a system tends to deteriorate as the software evolves along the years. Second, it is usual to have a lack of correspondence between the structure and the domain concepts of a system, which are often more related to the tasks necessary to design, develop, and maintain a system. This lack of correspondence leads to the scattering of concepts throughout the components of a system, which hinders developers to answer questions related to traceability, such as *where this conceptual feature is located at the source-code?* or *what features are realized by this piece of code?*

To address the need of different perspectives on software decomposition, Kersten and Murphy propose the *Task Context*, which is created by monitoring the tasks developers carry out during their development activities [1]. Therefore, this perspective captures a notion of decomposition that relates source-code entities to a *conceptual* task structure, which leads to an improvement on productivity [1], [2]. Unfortunately, this approach is not suitable to reverse engineering of system abstractions from an existing code base, since it collects information from the current interactions of the developer with the integrated development environment.

Another approach is to build a representation of the software from the code history. This is the approach followed by Zimmerman et al. [3], which deduces dependencies between software components from the common changes of source-code entities. These are *co-change dependencies*, which were further investigated in other research works [4]–[6]. In this way, Beyer and Noack [7], propose the use of *co-change dependencies* of coarse-grained software entities (i.e. object-oriented classes or interfaces) as input for building software clusters, and thus they coined the term *co-change clusters* to label the outcomes of their resulting perspective on software decomposition. Later, Silva et al. [8] empirically evaluate the use of coarse-grained co-change clusters as a software representation and found a mismatch between the cluster based decomposition and the modular structure of the target systems (in terms of Java packages) of four open-source systems.

Other works reason about the quality of a system decomposition using a notion of *conceptual cohesion*, which considers the vocabulary of terms present in software entities to estimate their semantic similarity [9]–[11]. Entities with high degree of conceptual cohesion might also be used to construct a different perspective of the software decomposition, named *semantic clusters* [11]. Accordingly, Santos et al. [12], investigated the use of *conceptual cohesion* and *semantic clustering* to assess remodularization.

It is important to note that the works mentioned above propose different perspectives of the software at the coarse-grained level, although the concepts of a software are often disperse at the fine-grained level. In this paper we investigate this issue using a new perspective of the software that is based on fine-grained co-change clusters. Therefore, the contributions of this paper are:

- We describe a framework for building a different perspective of a software (based on fine-grained co-change clusters) and a methodology to evaluate the conceptual cohesion of this software perspective (Section II).
- We report the results of an empirical assessment of our software perspective. In this analysis, we compute fine-grained co-change clusters for seven real-world systems and the observations reveal a higher cohesion

978-1-4673-9272-3/15 $31.00 © 2015 IEEE
DOI 10.1109/SBES.2015.16

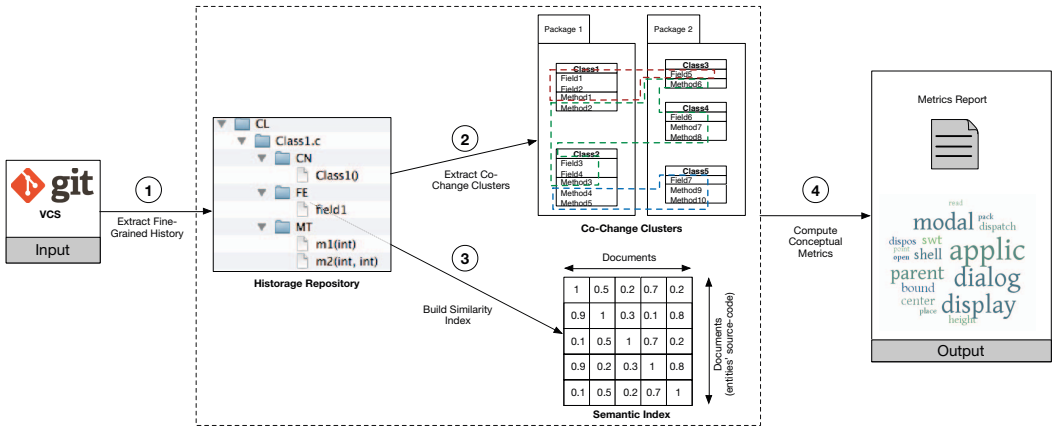


Fig. 1. Metrics Extraction Process. The numbered circles are the activities, which are executed in order.

of the resulting abstractions with respect to the typical decomposition of the systems (Sections III, IV, V).

Our findings has several implications. In particular, the terms related to the fine-grained entities that comprise a co-change cluster might serve as input to existing approaches that identify concerns from an initial setting [13]. We discuss some threats to our study in Section VI and relate our research to existing works in Section VII. Section VIII presents final remarks and future work.

II. Methodology

In this paper we aim at investigating the conceptual cohesion of co-change clusters. First we compute, for each target system, the co-change dependencies and respective co-change clusters using information that is available in Version Control Systems (VCSs) (first and second activities of Figure 1). We then compute similarity indexes, beginning from a list of frequent terms used in the source-code entities and computing the similarity between these entities using Latent Semantic Indexing (LSI) [14] (third activity on Figure 1). Finally, in the fourth activity of Figure 1 we compute a well defined metric for conceptual cohesion—as described in [10]. The conceptual cohesion metric is computed for the resulting co-change clusters and the original modular unities (packages and classes).

In the remainder of this section we present more details about each activity mentioned before. To enable the reproduction of our study, the scripts and data set we use are available on-line. [1]

A. Extracting Fine-Grained Version History

Typically, a VCS repository (such as GIT or SVN) contains the sequence of change sets applied to the software artifacts. In this paper we consider that a *commit* is a changeset that contains several artifacts, and that a fine-grained repository is a special kind of VCS that controls the history of changes applied to smaller software entities (e.g. classes, interfaces, attributes, methods, and constructors)[2].

Here we are particularly interested in the history of code constructs at the level of these smaller entities, and thus the goal of this first activity is to convert the original VCS repository of a system into a fine-grained repository. To this end, we use the *git2historage* tool [15] to convert a regular GIT repository into another GIT repository containing the history of the source-code at a fine-grained level—a *Historage Repository* (HR). Actually, *git2historage* transforms a conventional GIT repository into an HR containing the same number of the original commits. However, for each GIT commit, *git2historage* splits the related artifacts into a number of fine-grained entities. That is, the code related to each fine-grained entity is moved from the original source file to a new independent file.

The result of this first activity is illustrated as a tree layout of the file system on Figure 1, where the source-code of a class `Class1.c` is split on a number of files; one file for each source-code entity. From the HR repository, we build a detailed set of co-change clusters, as follows.

B. Extracting Co-Change Clusters

In general, the goal of software clustering is to discover groups of code entities considering some kind of mutual dependency and measure of similarity [16]. To apply a software clustering technique, it is first necessary to build a *Module Dependency Graph* (MDG)—a directed graph where: (a) the vertexes are source-code entities, and (b) the edges represent some kind of dependency. In this work we use Java classes and members as source-code entities; and the mutual dependency is based on the entities that had frequently changed together (in fact, the definition of *frequently* is subject to criteria that we will explore in Section III). There are two levels of granularity for co-change clusters: coarse-grained and fine-grained. *Coarse-grained co-change clusters* have classes as vertexes, and *Fine-grained co-change clusters* have members.

[1] http://github.com/mcesarhm/mpca and http://goo.gl/3E761S.

[2] In the remaining of this paper, for the sake of simplicity, when we refer to classes, we mean classes and interfaces. The same way, when we refer to members, we mean methods, attributes and constructors.

Here we use a specific kind of MDG: a co-change graph, that is formally defined as $G = (V, E)$, where V is a set of code entities and E is a set of pairs $(V_i, V_j) \in V \times V$, such that there is a least one commit that contains both V_i and V_j. In other words, V_i and V_j frequently change together. To improve the quality of the clusters, we followed several recommendations about how to build graphs from source-code history.

First, we do not use all changesets when building the graph; and thus we only consider commits explicitly related to at least one issue present in the *Issues Tracking System* (ISS) of the target systems—since this decision tends to produce clusters that are more semantically related [8]. With regard to the number of issues associated with commits, we found that only 4% of the commits are associated with more than one issue and only 0.7% are associated with more than 2 issues. For this reason, we might assume a small influence of tangled commits on the results [17], [18].

Second, we assume that a commit should not be considered in the analysis when the number of entities exceeds a given threshold [3]. Commits with too many entities are likely to contain unrelated code, thus we consider them as noise [19]. A code layout change is an example of evolution that might change many source-code files at once. In fact, as the commits must be associated to issues, we adapted this threshold to consider the number of entities per issue. Its specific value is further discussed in Sections III and IV. Finally, we also *prune* the graph using two metrics that measure the strength of co-change dependencies [20]: *support count*, which is the number of issues associated with both entities; and *confidence*, which measures the probability of a change in one entity when another changes. The graph will only contain edges with a minimum *support count* and a minimum *confidence*. The *confidence* metric is defined as

$$Confidence(V_i, V_j) = \frac{SupportCount(V_i, V_j)}{SupportCount(V_i, V_i)}$$

where the term $SupportCount(V_i, V_i)$ means the total of issues associated with V_i. Thus, $Confidence(V_i, V_j)$ is the proportion of issues where both V_i and V_j participates in relation to the total of issues that only V_i participates. Note that *Support Count* is symmetric—$SupportCount(V_i, V_j) = SupportCount(V_j, V_i)$, but *Confidence* is asymmetric—$Confidence(V_i, V_j) \neq Confidence(V_j, V_i)$.

So, when considering the criteria above, we simulated several threshold combinations to choose a suitable one, though we also considered some guidance from the literature as discussed further in Section III-B.

Therefore, to obtain a graph from the source-code, we iterate over the change history and their respective entities. Figure 2 illustrates this process, considering the source-code of two sample classes. Figure 2b also shows the commit log, from the corresponding HR, for the fine-grained entities contained in these classes. Note that the description of all commits contains a reference for some issue identifier. The column *Entities* are a simplified view of the files present in a specific commit. As we are illustrating a fine-grained cluster extraction, in this case the files affected by a commit represent methods. Figure 2c presents a first graph we build from our process, where the vertexes represent the methods and the edges represent the co-change dependencies between them. Each edge has a label in the form (S, C) where S is the *support count* and C is the *confidence*. The edges $m1 \rightarrow m4$ and $m4 \rightarrow m1$, for example, have labels $(3, 0.60)$ and $(3, 0.43)$ respectively. Note in Figure 2b that the method $m1$ participates in five commits, and the method $m4$ in seven. In addition, methods $m1$ and $m4$ participates together in three commits. Thus, the confidence values above are $3/5$ and $3/7$ respectively.

Further, according to the Figure 2, we prune that initial graph taking into account the quality criteria that leads to a definition of certain thresholds. In this example, we assumed *minimum support* equals 2 and *minimum confidence* equals 0.5. Figure 2d shows the graph with some dependencies removed. The remaining dependencies are those that comply with the *support count* and *confidence* thresholds.

Based on a co-change graph, there are several algorithms, methods, and tools for clustering software entities. In this work we use `Bunch`, an unsupervised cluster tool that applies a heuristic search algorithm based on random elements [21]. `Bunch` increases the reliability of the results by carrying out the clustering process several times. It receives an MDG as input and generates a hierarchical partition set (also called *Clustered Graph*, see Figure 2e). In our analysis, we configured `Bunch` with the *Agglomerative Clustering* action and the *Hill Climbing* clustering method. We choose this setup because it produces high quality results in predictable runtime [19]. Figure 2e shows the outcome produced by *Bunch* using the graph of Figure 2d as input.

C. Building the Similarity Index

Two steps are related to the activity of building the similarity index. The first step (preprocessing) uses the source-code as input, and outputs a term-document matrix, where terms are words collected from identifiers and comments, and documents are entities' source-code. This matrix is used as input for the second step and then discarded.

In more details, for each entity, we first obtain the last version of the artifacts from VCS and then extract terms from identifiers and comments. During preprocessing, we split identifiers that use camel case naming convention (e.g. for PrintingDevice we get Printing Device), or that is separated by underscores. This is the usual naming convention for identifiers in most popular languages, including *Java*, *C/C++*, and *C#*. We then proceed by removing *stop words*, words with only one character (to remove temporary or index variables), and words which occurs only once. Next, we reduce the words to their radical (this task is known as *stemming*). Finally, we use the *tf-idf* algorithm to give different weights to the frequent and infrequent terms, in order to compensate their influence [22]. Therefore, very frequent or infrequent words become less important in the computation of similarity index. Then, the original source-code of each entity is transformed into a corresponding bag of words from which we get a matrix with the terms as rows and the documents as columns, and cells representing the presence of terms in documents.

In the second step of this activity (that builds the *Similarity Index*), we use as input the term-document matrix from the first step and produce a document-document matrix, where

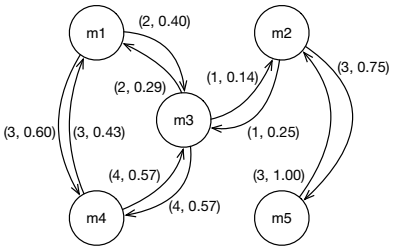
```
public class C1 {
    public void m1() { /* … */ }
    public void m2() { /* … */ }
}
```

```
public class C2 {
    public void m3() { /* … */ }
    public void m4() { /* … */ }
    public void m5() { /* … */ }
}
```

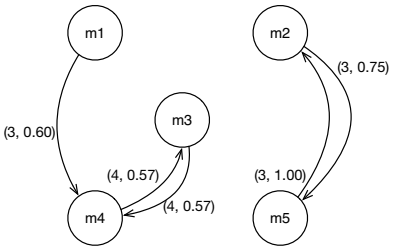
(a) Current source-code

Commit	Description	Entities
028a98d	Issue #1	m1, m3
d8fd425	Issue #2	m1, m3
c90c352	Issue #3	m1, m4
ad3f78a	Issue #4	m1, m4
cd5e305	Issue #5	m1, m4
7de2d7b	Issue #6	m3, m2
83850f6	Issue #7	m4, m3
59561f2	Issue #8	m4, m3
b8e3afd	Issue #9	m4, m3
3bed650	Issue #10	m4, m3
5afa3bb	Issue #11	m5, m2
121192e	Issue #12	m5, m2
44b80e9	Issue #13	m5, m2

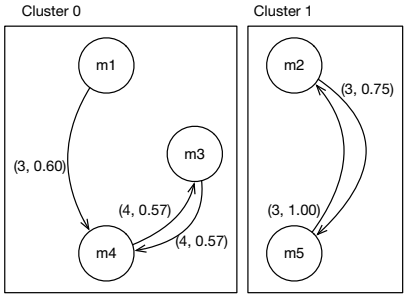
(b) Fine-grained commits



(c) Co-change graph



(d) Pruned co-change graph, using *minimum support* equals 2 and *minimum confidence* equals 0.5



(e) Co-change clusters

Fig. 2. Example of co-change cluster extraction (the edges' labels specify *support count* and *confidence* respectively.)

each cell has the index of similarity between two documents (see Figure 1). Here, documents are the entities' bag of words from the preprocessing. In fact, we build two matrices: one for coarse-grained entities, and one for fine-grained entities.

The similarity index is computed using LSI [14], that is an information retrieval technique for measuring the similarity between two documents, a document and a term, or two terms. According to this technique, the documents are modeled in a vector space considering the frequency of its terms. The similarity between two documents is determined by the cosine of the two corresponding vectors. As part of LSI, the dimensions of term-document matrix is reduced into a few orthogonal combinations, using a technique known as Singular Value Decomposition (SVD) [14]. The number of resulting dimensions is informed. When this technique is used for information retrieval, the resulting dimension is between 50–200. Differently, for source-code analysis, existing studies use a number between 20-50 [11].

D. Computing Conceptual Cohesion Metrics

Similar to other studies that compute the conceptual cohesion between software components [10], we also build the conceptual metrics using the vocabulary present in the source-code entities. The conceptual metrics are a collection of pairs (M, S), where the first element (M) represents a module and the second (S) corresponds to the *average similarity* of the source-code entities contained in the module. We consider two kinds of modules here: static and evolutionary. The static modules are further specialized into classes and packages; and the evolutionary modules into coarse-grained clusters and fine-grained clusters. As modular units, the coarse-grained clusters are equivalent to packages, and the fine-grained clusters are equivalent to classes. Thus, we can generically refer to both packages and coarse-grained clusters as coarse-grained modules; and, for both classes and fine-grained clusters as fine-grained modules.

To compute the similarity of the static modules (both coarse-grained and fine-grained), we retrieve from the HR the last version of the entities' source-code associated with each module. For each pair of entities of a static module, we retrieve their similarity index from the similarity matrix (that is built in the third activity of Figure 1). Next, we compute the module's similarity as the average similarity index of all possible pair of entities belonging to the module, as in [10]. Analogously, we consider the average similarity of the entire system as the average modules' similarity. Likewise, to compute the similarity of the co-change clusters, we iterate over the clusters and their respective entities (obtained in the second activity of Figure 1). We compute the cluster's similarity as the average similarity index of all possible pair of entities belonging to the cluster. Thus the average similarity of the entire system is the average of the similarity of its clusters.

Actually, we derive four metrics as the result of this activity. Assuming the similarity indexes of the entities in Figure 2 are given by the matrices of Figure 3, we can compute the following metrics:

- **Conceptual Cohesion of Packages (CCP)** given the package's classes, and the coarse-grained index, *CCP*

$$C = \begin{array}{c} \\ C1 \\ C2 \end{array} \begin{array}{c} \begin{array}{cc} C1 & C2 \end{array} \\ \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} \end{array}$$

(a) Coarse-grained index

$$F = \begin{array}{c} \\ m1 \\ m2 \\ m3 \\ m4 \\ m5 \end{array} \begin{array}{c} \begin{array}{ccccc} m1 & m2 & m3 & m4 & m5 \end{array} \\ \begin{pmatrix} 1 & 0.7 & 0.2 & 0.5 & 0.6 \\ 0.7 & 1 & 0.9 & 0.1 & 0.3 \\ 0.2 & 0.9 & 1 & 0.8 & 0.5 \\ 0.5 & 0.1 & 0.8 & 1 & 0.7 \\ 0.6 & 0.3 & 0.5 & 0.7 & 1 \end{pmatrix} \end{array}$$

(b) Fine-grained index

Fig. 3. Sample similarity indexes computed using LSI.

is the average similarity of all pairs of classes. For the entire system, it is the packages' average similarity. Given our sample data (Figure 3a), and assuming that $C1$ and $C2$ are declared within the same package, then $CCP = C[C1, C2] = 0.5$

- **Conceptual Cohesion of Classes (CCC)** given the class members, and the fine-grained index, *CCC* is the average similarity of all pairs of members. For the entire system, it is the classes' average similarity. Given our sample data (Figure 3b), $CCC = \frac{1}{2} \times ((F[m1, m2]) + \frac{1}{3} \times (F[m3, m4] + F[m3, m5] + F[m4, m5])) = \frac{1}{2} \times (0.7 + \frac{1}{3} \times (0.8 + 0.5 + 0.7)) = 0.68$
- **Conceptual Cohesion of Coarse-Grained Clusters (CGC)** given the cluster's classes, and the coarse-grained index, *CGC* is the average similarity of all pairs of classes. For the entire system, it is the clusters' average similarity. Given our sample data (Figure 3a), and assuming that $C1$ and $C2$ are within the same cluster, then $CGC = C[C1, C2] = 0.5$
- **Conceptual Cohesion of Fine-Grained Clusters (FGC:)** given the cluster's members, and the fine-grained index, *FGC* is the average similarity of all pairs of members. For the entire system, it is the clusters' average similarity. Given our sample data (Figure 3b), $FGC = \frac{1}{2} \times (\frac{1}{3} \times (F[m1, m3] + F[m1, m4] + F[m3, m4]) + (F[m2, m5])) = \frac{1}{2} \times (\frac{1}{3} \times (0.2 + 0.5 + 0.8) + 0.3) = 0.4$

III. Settings

This section brings details about the study settings we use in our investigation, discussing the target systems and the threshold selection we use to improve the quality of the co-change clusters.

This paper aims to investigate whether fine-grained co-change clusters present conceptual cohesion. It is important to understand if the clusters have been motivated by chance or if they are related to a set of related concepts. To this end, we use some metrics that have been already discussed in the literature [9]–[12].

A. Target Systems

We selected seven Java projects of different domains as the target systems of our study (Derby, Hadoop, Eclipse UI, Eclipse JDT, Geronimo, Lucene, and SIOP). These projects use either GIT or SVN as version control systems, and a significant number of their source-code commits are related to issues (ranging from 38% for SIOP to 98% for Hadoop). Six of the target systems are open source projects; while SIOP is one of the most important financial systems of the Brazilian Government, which the first author of this paper has contributed to [23]. All of them have large code bases and we could investigate at least two years of the development history of each system (see Table I for more details). Note that, due to some runtime constraints of `Bunch` regarding the size of the graph used as input [19], we had to limit the evolution history period of three projects (Hadoop, Eclipse UI, and Lucene). Nevertheless, we decided to use Bunch because [19] argues that it presents several advantages over other tools.

B. Selection of the Threshold Combination

We compute several clustered graphs based on different thresholds combinations. The set of thresholds we experiment are: *maximum number of entities per issue*, *minimum support count*, and *minimum confidence*. For the first threshold, we experiment with the value 50, as suggested by Beck and Diehl [19] and the value 100—because we grouped entities per issue instead of commit, and this scenario leads to a greater number of associations. For the second threshold, we experiment with the values 1 and 2, also following the recommendations of [8], [19]. Finally, for the third threshold, we experiment with the value 0 (according to Silva et al. [8]) and with the value 0.5 (a slightly more conservative scenario than the value 0.4 recommended in [19]).

The criteria we use to build the graphs discards some dependencies. As the co-change clusters contain only entities with dependencies between them, the set of entities contained in all clusters is a subset of all entities in a system. Therefore, there is a trade-off involving the use of more conservative thresholds in this case: although it might increase the quality of the clusters, it might also reduce the number of entities considered in the final analysis [19]. As a consequence, we discard a threshold combination when the ratio between the number of entities contained in clusters and the total number of entities of a system is below 1%. As we will discuss in Section IV, this ratio tend to be lower for certain target systems, and thus the threshold can not be too restrictive. Nevertheless, it is also necessary to discard graphs with a very low value for this ratio, to prevent from distorting the results.

The set of code entities were subdivided according to the level of granularity—coarse-grained entities, and fine-grained entities. There are eight possible threshold combinations that we use to build graphs. When considering each level, we have sixteen different clustered graphs. Some thresholds combinations applied to fine grained entities could not successfully run in `Bunch`, due to the huge number of vertexes of the graph. This is a known limitation of the tool [19]. Table II shows the clusters' cohesion per threshold combination. We also compute the similarity indexes in two situations: considering the source-code comments and not considering the source-code comments.

Given the cohesion of each combination the best is the one with greater value. For the results considering comments, *support count* with value 2 produced the best results regarding conceptual cohesion—85% of the combinations with this value were the best. Also, a *confidence* of 0.5 provided 79% of the best combinations. Finally, the *maximum entities per cluster* equals 100, is in 71% of the best combinations. Thus, we

TABLE I. BASIC DATA ABOUT TARGET SYSTEMS.

Name	Description	Version	Period of Analysis	KLOC	Packages	Classes	Commits	Commits Used
SIOP	Brazilian Planning and Budget System	1.26.0	2009/05/14-2014/05/15	521	241	5,611	12,061	100%
Derby	Relational database	10.11.1.1	2004/08/11-2014/09/10	679	215	3,252	6,656	100%
Hadoop	Large data sets processor	2.5.2	2013/01/01-2014/09/10	835	699	11,399	6,864	52%
Eclipse UI	Eclipse main UI	4.5M2	2011/01/01-2014/09/08	617	716	8,370	21,263	13%
JDT Core	Eclipse Java core tools	4.5M2	2001/06/05-2014/10/21	357	78	1,826	16,846	100%
Geronimo	Java EE application server	3.0.1	2006/08/24-2013/03/21	258	744	3,952	4,142	100%
Lucene	Text search engine	4.9.1	2011/01/01-2014/10/27	704	482	4,706	8,854	86%

TABLE II. TARGET SYSTEM'S CONCEPTUAL COHESION OF CO-CHANGE CLUSTERS. 'S' MEANS MINIMUM SUPPORT; 'C', MINIMUM CONFIDENCE; 'N': MAXIMUM ENTITIES PER ISSUE, 'CGC': COARSE-GRAINED CLUSTERS CONCEPTUAL COHESION, AND 'FGC': FINE-GRAINED CLUSTERS CONCEPTUAL COHESION. (BOLD NUMBERS SHOW THE SELECTED THRESHOLDS, '–' MEANS THAT THE COMBINATION DID NOT RUN IN BUNCH, '×' MEANS THAT THE RATIO OF ENTITIES IN CLUSTERS IS BELLOW 1%)

				SIOP		Derby		Hadoop		Eclipse UI		JDT		Geronimo		Lucene	
	S	C	N	CGC	FGC	CGC	FGC	CGC	FGC	CGC	FGC	CGC	FGC	CGC	FGC	CGC	FGC
with comments	1	0	50	0.41	0.39	0.37	–	0.39	–	0.54	–	0.24	–	0.44	0.36	0.47	0.31
	1	0	100	0.38	–	0.45	–	0.45	–	0.52	–	0.33	–	0.24	0.31	0.37	0.35
	1	0.5	50	0.40	0.14	0.54	–	0.47	–	0.55	**0.43**	0.45	–	0.44	0.16	0.48	0.42
	1	0.5	100	0.34	–	0.53	–	**0.58**	–	0.60	0.23	0.19	–	0.49	0.23	0.43	0.39
	2	0	50	0.19	0.31	0.44	**0.58**	0.47	0.24	0.42	×	0.56	0.69	0.55	×	0.52	0.43
	2	0	100	0.37	**0.42**	0.45	0.38	0.51	0.53	0.51	×	0.61	0.46	0.58	0.26	**0.58**	0.60
	2	0.5	50	0.27	0.35	0.31	0.48	0.52	0.61	**0.61**	×	0.62	0.51	0.58	×	0.56	**0.63**
	2	0.5	100	**0.52**	0.27	**0.62**	0.22	0.32	**0.62**	0.60	×	**0.65**	**0.71**	**0.59**	**0.49**	0.41	0.57
without comments	1	0	50	0.41	0.39	0.32	–	0.39	–	0.52	–	0.23	–	0.46	0.36	0.46	0.31
	1	0	100	0.38	–	0.42	–	0.44	–	0.49	–	0.31	–	0.22	0.31	0.36	0.35
	1	0.5	50	0.40	0.14	0.51	–	0.47	–	0.53	**0.43**	0.44	–	0.38	0.16	0.48	0.42
	1	0.5	100	0.34	–	0.51	–	**0.58**	–	**0.57**	0.24	0.17	–	0.52	0.23	0.42	0.39
	2	0	50	0.19	0.30	0.39	**0.58**	0.46	0.24	0.40	×	0.54	0.69	0.57	×	0.52	0.43
	2	0	100	0.37	**0.42**	0.41	0.37	0.50	0.53	0.49	×	0.59	0.46	0.55	0.26	**0.59**	0.60
	2	0.5	50	0.27	0.35	0.24	0.48	0.52	0.61	0.56	×	0.61	0.51	**0.59**	×	0.56	**0.63**
	2	0.5	100	**0.52**	0.27	**0.57**	0.22	0.30	**0.62**	0.57	×	**0.64**	**0.71**	0.57	**0.48**	0.40	0.57

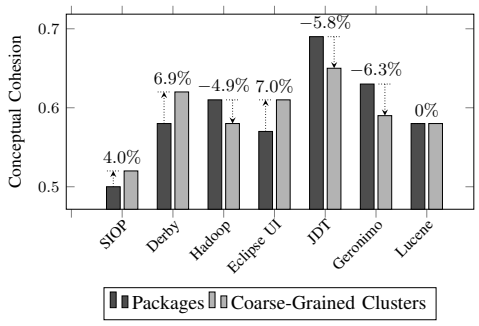
assume these values as the thresholds for *minimum support count*, *minimum confidence*, and *maximum entities per cluster*, respectively. It is important to emphasize that we use *maximum entities per cluster* equals 100, because this is less restrictive than the other considered option (50) that was first suggested in [24]. This can indicate that 50 is too restrictive, causing a loss of conceptual cohesion. The results without comments are near identical of the results with comments. Only 14% of the best combinations have different thresholds when ignoring comments. For this reason, in the remaining of this paper we will be only reporting results including comments.

IV. RESULTS

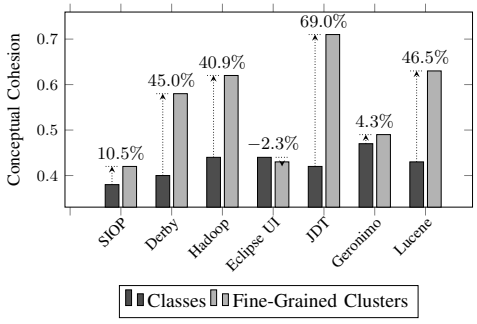
In this section we present the results of our research. For each target system we computed the average conceptual cohesion separately for packages, classes, and clusters. Figure 4 shows the average conceptual cohesion for each system (the metrics were introduced in Section II). Figure 4a shows data for the coarse-grained modules and Figure 4b shows the data for fine-grained modules. The labels near the top of the bars shows the growth (or decreasing) of the conceptual cohesion from the packages or classes to the clusters. Observing this numbers, we can see that, in general, the cohesion of clusters is greater than the cohesion of packages or classes. For coarse-grained clusters, only Hadoop, Eclipse JDT and Geronimo were significantly outperformed by packages, all of the rest of the clusters have a better conceptual cohesion than packages or classes. For fine-grained clusters, four systems have significant growth of conceptual cohesion, and only one had decreased (Eclipse UI). The other two systems (SIOP and Geronimo) had a low growth. As a future work, we will investigate if this divergence in growth is due to the original decomposition of each system.

Using the data from Figure 4, we can compute the average growth. For coarse-grained clusters, the average is 0.1 (standard deviation 5.9), and for fine-grained clusters, the average is 30.6 (standard deviation 26.5). Therefore, the co-change clusters, in the average, either maintains the same level of semantic when compared with the original organization of the systems (this is the case of coarse-grained clusters), or enhances the level of semantic (the case of fine-grained clusters). This also suggests that co-change clusters present conceptual meaning, and thus they are not a group of random entities that had changed together. This conclusion is particularly relevant for fine-grained clusters. Although the coarse-grained clusters represent a simple rearrangement of classes into new "packages", the fine-grained clusters represent a real different perspective of the software decomposition, as they break apart classes and form new abstractions with fine-grained entities—that are conceptually related. We believe that this new perspective might help in software maintenance and evolution, as we discuss in Section V.

Figure 5 shows the average number of entities (classes, members) in each corresponding module (packages, classes and clusters), for each system. Figure 5a shows the data for coarse-grained modules, and Figure 5b shows the data for fine-grained modules. Here, we consider that the module of

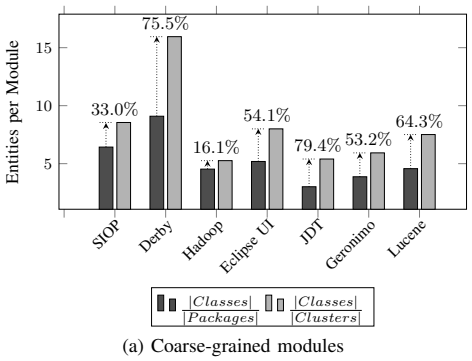


(a) Coarse-grained modules cohesion

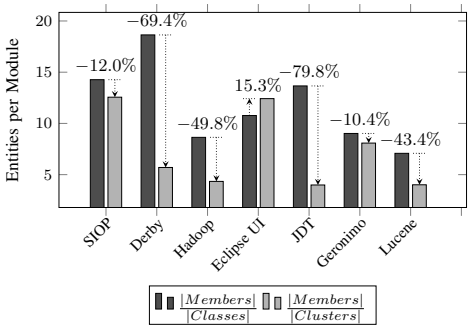


(b) Fine-grained modules cohesion

Fig. 4. Target System's Conceptual Cohesion.



(a) Coarse-grained modules



(b) Fine-grained modules

Fig. 5. Proportion of entities in relation to modules.

a class is either a package or a coarse-grained cluster, and the module of a member is either a class or fine-grained cluster. From that data, we can compute the average growth of entities per module. For coarse-grained modules, the average is 53.7 (std.dev. is 22.7), and for fine-grained modules, the average is -34.3 (std.dev. is 32.7). Comparing these numbers, we can see two different trends: while the number of classes per package tend to be smaller than the number of classes per coarse-grained clusters, the number of members in classes tend to be greater than the number of members in clusters. With regard to the average clusters' size, we have 7.3 (std.dev. is 3.82) for fine-grained and 8.09 (std.dev. is 3.7) for coarse-grained.

We also calculate the correlation (Pearson coefficient) of the growth of conceptual cohesion with the growth of entities per module. In the coarse-grained case, we realize a small correlation between the conceptual cohesion growth and the number of entities per module growth (that is, a correlation value of 0.135). Differently, in the fine-grained case, we realize a *strong and inverse correlation* between the conceptual cohesion growth and the number of entities per module growth (a correlation value of -0.945). Thus, we conjecture that the original classes deal with non-cohesive responsibilities, and that the fine-grained clusters captured their different concepts and form new "classes" that are more conceptually related than the original ones. In that sense, a reverse engineering approach using fine-grained clusters might also lead to a new perspective which locate more easily concepts throughout different classes of a system (and therefore reduces the scattering of concepts), and, thus, can be used to better understanding a software.

It is important to note that we are not suggesting a general restructuring of the software decomposition in terms of co-change clusters. Instead, these clusters provide an orthogonal perspective of the software organization that presents significant improvement of conceptual cohesion (in the fine-grained case). In addition, it is important to note that the computation of co-change clusters discards many dependencies due to the pruning criteria we use. Accordingly, many code entities are also discarded—since the resulting clusters only include entities with at least one co-change dependency to another entity.

We also investigate the effect of pruning in the proportion of entities preserved in the clusters in relation to the original number of entities in a system (Table III shows the results of this investigation). The column #C/#OC relates to the coarse-grained clusters, and the column #M/#OM relates to the fine-grained clusters. We can see in Table III that, on average, the preservation is lower in the case of fine-grained clusters. However, if we consider only the number of entities changed in the period used for computing graphs, these numbers for Eclipse UI raise to 11.02% and 23.79%, (for coarse-grained and fine-grained, respectively); and for Hadoop they raise to 53.1% and 6.06%. The smaller improvement on Hadoop might be due to the fact that the period length used for building the graph is closer to the period length of the whole history, compared with the respective periods of Eclipse UI. Thus, for these target systems, the entity preservation is greater for entities that had recently changed. From a software evolution perspective, these entities are the focus of our interest, because they are more susceptible to change in a near future.

The numbers we discussed above are significant, since in the existing literature about software co-change clusters, a few works brings references about the smaller number of

TABLE III. PROPORTION OF ENTITIES PRESERVED BY THE CLUSTERING PROCESS. #C=NUMBER OF CLASSES, #M=NUMBER OF MEMBERS, #OC=ORIGINAL NUMBER OF CLASSES, #OM=ORIGINAL NUMBER OF MEMBERS

System	#C/#OC (%)	#M/#OM (%)
SIOP	31.10	5.25
Derby	45.79	4.04
Hadoop	29.71	2.71
Eclipse UI	5.83	6.16
Eclipse JDT	20.42	5.34
Geronimo	15.16	1.95
Lucene	32.58	4.03
Average	25.80	4.21
Std.Dev.	13.10	1.51

entities preserved after building co-change graphs (for further discussion in that subject, see [19]). Figure 6 shows that, on average, entities contained in clusters are more frequently changed than the whole set of entities in a system. More specifically, the entities that pertain to at least one co-change cluster, were changed 25% more times than the average of changes for all entities. This data reveals quantitatively that the entities within co-change clusters are more relevant from the evolutionary perspective.

Nevertheless, in the case of this research, this small proportion of fine-grained entities in clusters can be viewed under another perspective: the total number of lines of code they represent. Hence, we computed this measure of size for the entire set of fine-grained clusters for the target systems. We found that, on the average, those clusters represent 31.4 KLOC with a standard deviation of 20 KLOC. Therefore, the size of the fine-grained clusters that present high cohesion could be compared to medium size systems.

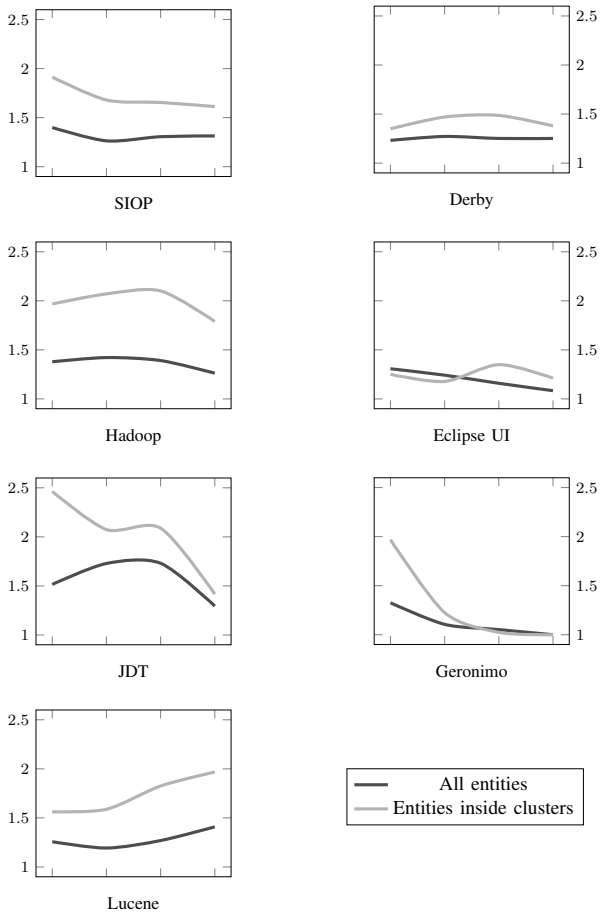


Fig. 6. Average of commits per entity. X axes represent the last two years of change history for each system, from past (left) to present (right).

V. IMPLICATIONS OF OUR RESULTS

In this section we discuss the implications of the main finding of this paper: that is, fine-grained co-change clusters present a high degree of conceptual cohesion—where the notion of conceptual cohesion relates to the vocabulary present in source-code elements [10]. Therefore, the fine-grained co-change clusters provide a complementary view of a high cohesive modular decomposition. In particular, those clusters might help the identification of concepts that scatter throughout different classes. In that sense, the vocabulary which represents the semantic of a fine-grained cluster can be viewed as a kind of code annotation, that might allow *virtual separation of concerns* [25]. Also, the coverage of fine-grained clusters in relation to code entities can be expanded by applying specific techniques, such as the approach proposed by Nunes et al. [13]. This way, the mapping between the concepts embedded in the fine-grained clusters might serve as an initial *seed* to relate features and source-code. Additionally, each co-change cluster have a list of terms semantically related that can be used to infer the concepts associated to a particular cluster, or to search certain clusters associated with a query expressed as a list of correlated terms. Again, this can aid feature identification or location, and we envision its use in conjunction with search based tools for software maintenance.

VI. THREATS TO VALIDITY

In this section we present a discussion about some questions that might threat the validity of our work. We organize them according to the internal, construct, and external threats.

Internal Validity. We applied the same method to all target systems, including thresholds. However, we cannot ensure that some combination of thresholds favor or disfavor a particular project. Specifically, the effect of the threshold used to limit the maximum entities per commit depends on the development process of the subject project. To minimize these effects, we chose the thresholds according to the guidance of previous studies. When searching for the best combination of thresholds, in regard to semantic similarity, on the majority of projects the same combination was selected as the best. As the co-change clusters contain only a fraction of the original set of code entities, it is possible that certain portions of one project would produce metrics more favorable than of another project. But, as the results show a clear trend among most systems, for the computed metrics, probably the effect of this threat is not significant. The number of clusters generated by `Bunch` depend on its algorithm, and thus it can interfere on the metrics values as well. Our results showed that there is a strong

inverse correlation between the clusters growth (influenced by the number of clusters), and their conceptual cohesion (see Section IV). Thus, the effect of this threat is not significant, because this correlation is consistent in all systems.

Construct Validity. While we require an association between issues and commits to raise the semantic relation of the clusters, the precision of this association cannot be ensured. That is, we can not verify whether a given commit is related to the real issues that motivated the software revision. In addition, this association reduces the number of commits considered in our analysis. This reduction depends on the proportion of commits associated to issues in each system, as show in Section III. Also, we have to constrain the history period when building the graphs for some projects, due to a `Bunch` limitation. But the potential impact of this threat does not invalidate our results, according to the analysis in Section IV. This analysis show that, if we analyze only the constrained period, the results will be even more favorable to the fine-grained co-change clusters. The representation of the graph was in conformance with existing works, albeit there are alternatives [26]. Our approach has some differences from the majority of studies. In particular, we use fine-grained entities as elements; and the issues in common as dependency criteria instead of commits.

External Validity. We selected a small set of *Java* projects for this study. This can potentially limit the generalization of our results. Nevertheless, these projects have been used in previous research works, and we choose a wide range of applications, not limited to open-source ones. All projects had large codebases with a long history of changes. As our purpose is to disclosure the conceptual cohesion of co-change clusters, we expect that the the methodology we present in this paper can be reproduced in other projects. The small number of preserved entities in fine-grained clusters can threat the validity of the results for this level of granularity. Nevertheless, even this small set of entities can represent latent concepts inside codebases, and, as such, can be expanded to discover more code fragments related to the same concept [13], and, thus, raise its usefulness. We used `Bunch` as the only tool for software clustering. The choice for `Bunch` was made after a broad research on the literature, and it was found among the tools that produce the better results for software clustering [27]. Also, the `Bunch`'s limitations were mitigated according to the previous explanations.

VII. Related Work

The work of Gall et al. [28] was the first to explore the information from version history repositories to detect evolutionary coupling between modules. Zimmermann et al. [20] proposed an approach to determine evolutionary coupling between fine-grained entities, used for predicting further changes [3]. Beyer and Noack [26] introduced the use of co-change dependencies in clustering, while Vanya et al. [29] proposed a semi-automatic approach to suggest remodularization. In their approach, given an initial partition, inter-partitions evolutionary dependencies are identified. Silva et al. [8] propose an approach to assess modularity using co-change clusters. Their method use the Chameleon [30] tool to cluster coarse-grained entities that are compared with the actual package decompositions. According to their work, mismatches between the co-change clusters and the package decomposition can suggest new directions for restructuring the package hierarchy.

Regarding the semantic assessment of source-code, Maletic and Marcus [9] introduced the use of LSI to extract semantic information code entities. Marcus and Poshyvanyk [10] proposed new measures for class cohesion based on LSI. These measures, though conceptual, are correlated with traditional software cohesion measures. Their metric (*Conceptual Cohesion of Classes (C3)*) computes the average similarity index between each pair of methods of a class. For each method, a document is built, containing the terms extracted from identifiers and comments. Kuhn et al. [11], introduced *Semantic Clustering*, a technique based on LSI to group source-code artifacts that use the same vocabulary. To enable clustering, they use a similarity metric based on C3, generalizing it in order to compute the similarity of a cluster. In this case, the metric is defined as the average C3 of the classes contained in the cluster.

Bavota et al. [31] proposed the use of semantic information of the source-code, combined with its structural information to recommend remodularization. Santos et al. [12] experimented with semantic clustering also to evaluate software remodularizations. Their approach compares conceptual metrics values between a number of versions of a system, to analyze the relationship between the remodularizations promoted by the new versions and the semantic clusters. They also propose new metrics for conceptual cohesion of clusters and packages, that are the average cosine similarity between each pair of classes in clusters ans packages, respectively. Dit et al. [32] proposes an approach to measuring the textual coherence of the user comments in bug reports using Latent Semantic Analysis (LSA). They define the semantic similarity of a bug report as the average cosine similarity of the comments, and the coherence of a bug report as the semantic similarity. Silva et al. [8] use a similar approach, but they compute the similarity of the issues associated with the clusters using only the issues' short description, instead of the whole issue report.

This paper is different from the aforementioned works in several aspects. First, our approach uses fine-grained entities and thus it increases the software cohesion of the resulting clusters, when compared to coarse-grained alternatives. In particular, we differ from the work of Kuhn et al. [11] because they compute the clusters based on the semantic similarity between classes. Thus, there are two main differences in this case: we compute the clusters based on the co-change dependencies; and we use finer-grained source-code entities instead of only classes as entities. Our intention is to verify the conceptual cohesion of co-change clusters, so we are not interested in proposing a new clustering method. Also, the use of fine-grained entities allows us to discover interesting conceptual properties of the co-change clusters based on them, with results much more significant than for the coarse-grained clusters. In relation to the work of Marcus and Poshyvanyk [10], our work uses an extended version of their metrics. Although their work concentrates on analyzing the merit of the proposed metric (C3), our work extends the original definition and we do not compare its performance with another equivalent cohesion metric.

VIII. Conclusion

Software clustering use coupling information involving code entities to group entities with strong dependencies. Recent works have experiment with different kinds of software clustering techniques, including structural, dynamic, semantic, and based on change history; also with different levels of code entities granularity, but particularly with coarse-grained ones (like source-code files, C++ namespaces, Java packages, and object-oriented classes). In this paper we presented a novel perspective of the software decomposition, based on fine-grained co-change clusters. We also investigated the conceptual cohesion of the resulting decomposition perspective, using a well-known measure of semantic similarity. Our evaluation considered seven real target systems, six open-source projects and one system from the financial domain of the Brazilian Government. The main conclusion is that *fine-grained co-change clusters* present a higher degree of conceptual cohesion when compared to the typical decomposition of the systems (based on the package hierarchy) and to another perspective based on *coarse-grained co-change clusters*. Therefore, our new perspective of the software decomposition, though limited in the number of entities, present a cohesive vocabulary associated to each cluster, allowing the discovery of concepts scattered on a number of code entities. As a future works, we aim at investigating the hypothesis that combining our perspective with existing techniques for feature expansion would help on the identification of features during reverse engineering activities.

References

[1] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 1–11.

[2] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić, "The emergent structure of development tasks," in *ECOOP 2005-Object-Oriented Programming*. Springer, 2005, pp. 33–48.

[3] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 429–445, 2005.

[4] H. Gall, M. Jazayeri, and J. Krajewski, "Cvs release history data for detecting logical couplings," in *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, Sept 2003, pp. 13–23.

[5] T. Zimmermann and P. Weißgerber, "Preprocessing cvs data for fine-grained analysis," in *Proceedings of the First International Workshop on Mining Software Repositories*. sn, 2004, pp. 2–6.

[6] A. Hassan and R. Holt, "Predicting change propagation in software systems," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, Sept 2004, pp. 284–293.

[7] D. Beyer and A. Noack, "Mining co-change clusters from version repositories," Tech. Rep., 2005.

[8] L. L. Silva, M. T. Valente, and M. d. A. Maia, "Assessing modularity using co-change clusters," in *Proceedings of the of the 13th international conference on Modularity*. ACM, 2014, pp. 49–60.

[9] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 103–112.

[10] A. Marcus and D. Poshyvanyk, "The conceptual cohesion of classes," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 133–142.

[11] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[12] G. Santos, M. T. Valente, and N. Anquetil, "Remodularization analysis using semantic clustering," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 224–233.

[13] C. Nunes, A. Garcia, C. Lucena, and J. Lee, "Heuristic expansion of feature mappings in evolving program families," *Software: Practice and Experience*, 2013.

[14] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *JAsIs*, vol. 41, no. 6, pp. 391–407, 1990.

[15] H. Hata, O. Mizuno, and T. Kikuno, "Historage: fine-grained version control system for java," in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*. ACM, 2011, pp. 96–100.

[16] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization," in *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*. IEEE, 1997, pp. 33–43.

[17] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 2013, pp. 121–130.

[18] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 341–350.

[19] F. Beck and S. Diehl, "On the impact of software evolution on software clustering," *Empirical Software Engineering*, vol. 18, no. 5, pp. 970–1004, 2013.

[20] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*. IEEE, 2003, pp. 73–83.

[21] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *Software Engineering, IEEE Transactions on*, vol. 32, no. 3, pp. 193–208, 2006.

[22] S. T. Dumais, "Improving the retrieval of information from external sources," *Behavior Research Methods, Instruments, & Computers*, vol. 23, no. 2, pp. 229–236, 1991.

[23] "Siop: Citizen service letter," http://www.orcamentofederal.gov.br/biblioteca/cartas-de-servico/carta_de_servicos_SIOP.pdf.

[24] F. Beck and S. Diehl, "Evaluating the impact of software evolution on software clustering," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 99–108.

[25] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 311–320.

[26] D. Beyer and A. Noack, "Clustering software artifacts based on frequent common changes," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. IEEE, 2005, pp. 259–268.

[27] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *Software Engineering, IEEE Transactions on*, vol. 33, no. 11, pp. 759–780, 2007.

[28] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 190–198.

[29] A. Vanya, L. Hofland, S. Klusener, P. Van De Laar, and H. Van Vliet, "Assessing software archives with evolutionary clusters," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 192–201.

[30] G. Karypis, E.-H. Han, and V. Kumar, "Chameleon: Hierarchical clustering using dynamic modeling," *Computer*, vol. 32, no. 8, pp. 68–75, 1999.

[31] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 1, p. 4, 2014.

[32] B. Dit, D. Poshyvanyk, and A. Marcus, "Measuring the semantic similarity of comments in bug reports," *Proc. of 1st STSM*, vol. 8, 2008.