Feature-to-Code Traceability in a Collection of Software Variants: Combining Formal Concept Analysis and Information Retrieval

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai and Christophe Dony
LIRMM Laboratory, Montpellier, France
{Eyalsalman, Seriai, Dony}@lirmm.fr

Abstract

Today, developing new software variant to meet new demands of customers by ad-hoc copying of already existing variants of a software system is a frequent phenomenon in the software industry. Typically, maintaining such variants becomes difficult and expensive over the time. To re-engineer such software variants into a software product line (SPL) for systematic reuse, it is important to identify source code elements that implement a specific feature in order to understand product variants code. Information Retrieval(IR) methods have been used widely to support this purpose in a single software. This paper proposes a new approach to improve the performance of IR methods in a collection of similar software variants. Our proposal produces following two improvements. First, increasing the accuracy of IR results by exploiting commonality and variability across software variants. Secondly, increasing the number of retrieved links that are relevant by reducing the abstraction gap between feature and source code levels. We have validated our approach with a set of variants of two different systems. The experimental results showed that the proposed approach outperforms the conventional application of IR as well as the most relevant work on the subject.

1. Introduction

Today, developing new software variants to meet new demands of customers by ad-hoc copying of already existing variants of a software system is a frequent phenomenon in the software industry. This is because that software companies need to develop new software products with major improvements during a short time to remain competitive in their respective market. When the number of such software variants growths, the need for a systematic reuse strategy becomes apparent because maintaining such variants becomes more difficult and expensive over the time [1]. In contrast to such ad-hoc reuse mechanism, software product line engineering (SPLE) provides a systematic strategy for reuse. It produces a short time-to-market products in a cost-efficient way by achieving large-scale software reuse. It often exploits available software artifacts of existing legacy systems to build SPL's core assets (e.g.,source code, design documents, features and so on) [2]. A feature is a prominent or distinctive user-visible aspect, quality or characteristic of a software system [3].

To re-engineer legacy software variants into an SPL for systematic reuse, it is important to identify source code elements (e.g., classes) that implement a specific feature. This identification is known as traceability links or feature location. Traceability links in legacy systems are particularly important and helpful to understand software variant code. Additionally, traceability links in SPLE are needed to derive concrete product from SPL's core assets by selecting features and their respective source code elements [4]. Information Retrieval (IR) methods are used widely to automate traceability links recovery for single software product [5]. The conventional application of IR methods consists in lexical matching of all features (i.e. their descriptions) for a single software to its entire source code information. These features and the associated source code are called IR spaces.

In this paper, we improve the performance of IR methods when they are applied to identify feature-to-code traceability in a collection of object-oriented software variants. This work produces following two improvements. First, the proposed approach enhances the accuracy of IR results by reducing the number of false positive links. This is achieved by reducing IR spaces (i.e., feature and source code spaces) trough exploiting commonality and variability across software variants. Second, it increases the number of retrieved links that are relevant by bridging the abstraction gap between feature and source code levels. This bridging is performed by introducing an intermediate level called "code-topic". A *code-topic* is a cluster of similar classes that are grouped together to cover the same topic. Such code-topics can be a functionalities implemented by the source code. Using *code-topic* we can get enough information (e.g., classes' identifiers) to textually match these classes

together with a feature description where we consider that a feature is a bundle of functionalities. In this paper, we investigate the results of identifying *code-topics* based on textual information as well as combining textual and structural information.

We use separately Formal Concept Analysis (FCA) and lexical similarity computing to reduce IR spaces. It also combine FCA, structural and textual information to derive *code-topics*. Traceability links between features and their corresponding *code-topics* are recovered using LSI. Then, each feature are linked to their classes by decomposing each *code-topic* to its classes

The rest of this paper is structured as follows. Section II and III shows background and the traceability recovery process. Section IV shows experimental results and evaluation. Section V discusses related works. Finally, section VI concludes our work.

2 Background

2.1 Assumptions

For the purpose of this work, we focus on functional features that express the behavior or the way users may interact with a product [6]. Typically legacy software variants implement a set of functional features either common, shared among all variants, or optional, shared among some variants. We restrict ourselves to object-oriented systems. In an object-oriented source code, the functional features can be implemented by a set of packages, classes, methods and attributes. As the class represents a main building unit in all object-oriented languages and most often developers think about the class as a set of responsibilities that simulate a concept or functionality from of the application domain [7]; we assume that the functional feature is implemented by a set of classes.

Intuitively, the functional feature is a bundle of functionalities that are related. Thus, we propose the *code-topic*, as a cluster of similar classes that are grouped based on either lexical similarity or combing lexical and structural similarity to implement a functionality. In this work, we investigate the results of two methods to derive *code-topics*. *Code-topics* constitute an intermediate level that bridges the abstraction gap between feature and source code levels. Features provide an abstraction of requirements and their implementations are scattered over multiple classes. Consequently, a feature (i.e., its description) can be matched to a set of *code-topic* (i.e., their source code information) representing its functionalities. This allows us to easily map a feature to a set of classes that are similar and grouped as a *code-topic* instead of mapping each feature to each class separately.

2.2 Textual Information and IR

Textual information in source code refers to identifier names and internal comments. This information records important domain knowledge about a software system. Information Retrieval (IR) exploits this information to locate a feature's implementation. IR works by lexical matching a set of textual artifacts with a query and ranking these artifacts against the query. Different IR methods such VSM and LSI have been proposed. However, they share four steps to locate a feature in the source code [8]. First, IR methods start by building a collection of documents called corpus. A document is a list of source code information (i.e., identifiers and comments) found in a named block of source code such as a method, class or package. Second, the corpus undergoes a preprocessing step. A preprocessing involves normalizing the documents content such as stop word removal and stemming. In the third step, a term-by-document matrix is created by using the corpus. The matrix's columns correspond to the corpus documents and rows represent terms that are extracted from these documents. The matrix values indicate the number of occurrences of a term in a document according to a specific weighting scheme. Finally, a user make a query that describes specific feature to be located. Each query is manipulated by the same preprocessing techniques as the corpus.

In VSM, documents are represented by vectors of terms so that each column in the term-by-document matrix is a document vector of terms that appear in all documents. LSI differs from VSM by using a Singular Value Decomposition (SVD) technique which is used to overcome the synonymy and polysemy issues. The SVD divides the term-by-document matrix to create LSI subspaces. In the LSI subspaces each document has a corresponding vector. We use the vector representation to compute similarity. In both LSI and VSM, the textual similarity between documents and queries is measured by the cosine of the angle between their corresponding vectors [9].

The effectiveness of IR methods is commonly measured by their precision, recall and F-measure. For a given query, precision is the percentage of retrieved links that are relevant to the total number of retrieved links. Recall is the percentage of retrieved links that are relevant to the total number of relevant links. F-measure makes a trade-off between precision and recall so that it gives a high value only in the case that both recall and precision values are high. All measures have values in a range [0, 1]. Higher precision, recall and F-measure mean better results [8].

2.3 Structural Information

Structural information refers to the dependency relationships in software's source code such as method calls and

inheritance relationships. These relationships are important for feature-to-code traceability because a feature's implementation spans multiple classes and these classes that participate to implement a feature are linked. For example, In compositional approaches such as feature-oriented programming, the code that implements a specific feature is encapsulated as a cohesive unit [10]. Thus, we use cohesion metrics in order to measure the degree to which classes are linked to each other. We rely on four cohesion measures which capture different types of interactions among classes These measures include:

1. **Inheritance relationship:** When a class inherits resources of another class.
2. **Method call:** When methods of one class use methods of another class.
3. **Shared method invocation:** when two methods of two different classes have a shared method invocation.
4. **Shared attribute access:** when two methods of two different classes have a shared attribute access.

2.4 Fundamentals of Formal Concept Analysis

Formal Concept Analysis (FCA) is a technique for data analysis and knowledge representation based on lattice theory. It identifies meaningful groups of objects sharing common attributes and provides a theoretical model to analyze hierarchies of these groups. The main goal of FCA is to define a concept as a unit of two parts [11]: extent and intent. The extent of a concept is the objects covered by the concept, while the intent is the attributes, which are shared by all the objects covered by the concept [11].

In order to apply FCA, the formal context or incidence table of objects and their attributes is needed Formally, the formal context is defined as a triple $K = (O, A, R)$ where O and A are sets of objects and attributes respectively and R is a binary relation between objects and attributes, indicating which attributes are possessed by each object, i.e., $R \subseteq O \times A$. For a given formal context K, a formal concept is a pair (E, I) composed of an object set $E \subseteq O$ and an attribute set $I \subseteq A$. $E = \{o \in O | \forall a \in I, (o, a) \in R\}$ is the extent of the concept. $I = \{a \in A | \forall o \in E, (o, a) \in R\}$ is the intent of the concept. The set of all concepts of a formal context constitutes a concept lattice. There are several algorithms to compute concept lattices from a given formal context. In this work, we depend on Galois lattices that ignore empty concepts.

3 The Traceability Recovery Process

This section describes input data and steps for our feature-to-code traceability process. This process takes two

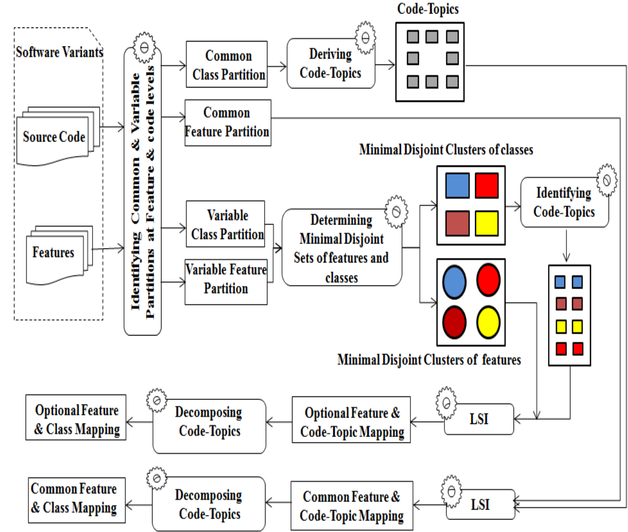

Figure 1. An overview of our approach.

inputs: object-oriented source code and feature descriptions of a set of software variants. Each feature is identified by its name and the description which consists of a short paragraph. This information about features is generally available due to the need for product customization.

Figure 2 shows the traceability recovery process which consists of four main steps. The first step aims to reduce LSI spaces. Lexical similarity computing divides all features and classes of a given set of software variants into common and variable partitions. At the feature level, common and variable partitions consist of all common and optional features across software variants respectively. At the source code level, common and variable partitions consist of classes that implement common and optional features respectively. Also in the first step, the variable partitions are fragmented into minimal disjoint sets using FCA. Second, *code-topics* are derived from common class partition and each minimal disjoint set of classes that are computed in the previous step. In the third step, the traceability links between features and their possible corresponding *code-topics* are established using LSI. Finally, by determining *code-topics* related to each feature, we can easily determine classes that implement each feature by decomposing each *code-topic* to its classes.

3.1 An Illustrative Example

As an illustrative example through this paper, we consider four variants of a bank software. *Bank_V1.0* supports just core features for any bank software: `CreateAccount`, `Deposit`, `Withdraw` and `Loan`. *Bank_V1.1* has, in addition to the core features, `OnlineBank`, `Transfer` and `MobileBank` fea-

Table 1. Formal context for describing bank systems differences.

	OnlineBank	Transfer	Consortium	BillPayment	Conversion	MobileBank
$V1.2 - V1.0$	X		X	X	X	
$V1.0 - V2.0$						
$V2.0 - V1.0$	X	X	X	X	X	X
$V1.1 - V1.2$		X				X
$V1.2 \cap V2.0$	X		X	X	X	
...						

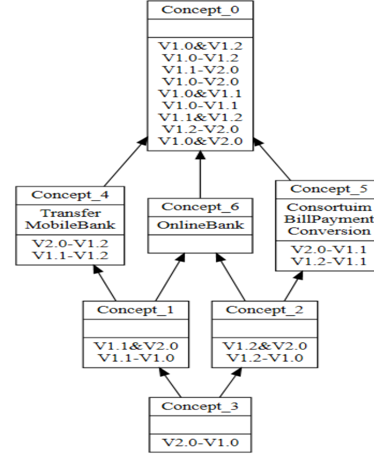

Figure 2. The concept lattice for the formal context of Table I.

tures. *Bank_V1.2* supports not only core features but also new features: `OnlineBank`, `Conversion`, `Consortium` and `BillPayment`. *Bank_V2.0* is an advanced application. It supports all previous features together.

3.2 Reducing LSI spaces

3.2.1 Determining common and variable partitions at feature and source code levels

At the feature level, we rely on lexical similarity of feature names and their descriptions to determine the common partition by identifying two subsets of common features. First, we define a subset of those features that have the same name across software variants. Second, a subset of features that have the same description but may have different names due to changes in software environment or to the adoption of different technology. We compute the longest common subsequence of feature description terms to determine the second subset. We consider two features identical if and only if they have the same subsequence terms of their description. In our illustrative example, all core features form the common partition at feature level. The remaining features (i.e., optional features) in each variant form together the variable partition at the feature level.

At the source level, we need to divide the source code of a given set of software variants into a common and variable partition. To do so, we represent the source code for each variant as a set of Elementary Construction Units (ECUs). Each ECU has the following format: $ECU = PackageName_ClassName$

Each software variant P_i is abstracted as a set of *ECUs* as follows: P_i={$ECU_1, ECU_2, \ldots, ECU$n }. An *ECU* reveals any changes at package and class levels (e.g., adding or removing packages or classes in a subsequent variant). These changes co-occur with adding or removing features in a subsequent variant. Common `ECUs` shared by all software variants represent common classes composing the common partition at source code level. The variable partition at source code level is composed of the remaining classes in each variant. The Common *ECUs* are computed by conducting a lexical matching among *ECUs* for all variants where we assume that developers use the same vocabulary to name source code identifiers.

3.2.2 Fragmentation of the variable partitions into minimal disjoint sets

To reduce further the space related to the variable partitions, computed in the previous, our approach fragments these partitions into minimal disjoint sets of optional features and their respective minimal disjoint sets of classes.

Minimal disjoint sets of optional features are computed based on FCA. We apply FCA on all variant-differences. For two software variants P_1 and P_2, we create three variant-differences. $P_1 \bigcap P_2$ (a set that contains all the optional features that P_1 and P_2 have in common), $P_1 - P_2$ (a set that contains features existing only in P_1 but not in P_2) and $P_2 - P_1$ (a set that contains features existing only in P_2 but not in P_1). The variant-differences aim at identifying differences between each pair of variants at feature level taking into account all combinations between software variants. As an example, if we consider variants $Bank_V1.2(V1.2)$ and $Bank_V2.0(V2.0)$ of our illustrative example, three variant-differences can be created as follows: $V1.2 - V2.0$ = $\{\phi\}$, $V2.0 - V1.2$ = {*OnlineBank, Conversion, Consortium, BillPayment*} and $V1.1 \bigcap V1.2$ = {*Transfer, MobileBank*}. This way to compute variant differences is similar to the one proposed by Xue et al. [12].

We use FCA to group optional features via concept lattice into minimal disjoint sets. To do so, we define the formal context of FCA as follows: all variant-differences and optional features represent objects (extent) and at-

tributes (intent) respectively. A relation between a variant-difference and an optional feature indicates to an optional feature possessed by a variant-difference. Table 1 shows a formal context related to our illustrative example. Figure 3 shows the concept lattice related to the formal context of Table 1. Each node in the lattice represents a concept having features as intent and these features are shared by variant-differences which constitute the associated extent. We are interested in the concepts associated with a set of optional features (such as the *Concept_5* in Figure 3). They allow us to know how to obtain minimal disjoint sets of optional features and determine corresponding classes implementing these features.

The minimal disjoint set of classes related to each minimal disjoint set of features is calculated as follows. For a given set of optional features (i.e., a concept computed by FCA), we analyze the concept's extent to determine which variants should be compared. We encounter two cases to compute a relevant set of classes. First, if the concept's extent is not empty, we randomly select only one variant-difference from the variant-differences listed in its extent. For instance, considering $Concept_5$ in Figure 3 , we can select the first variant-difference ($V2.0 - V1.1$) to identify a set of classes that are present in $Bank_V2.0$ but absent in $Bank_V1.1$. The resulting set of classes implements the features located in the $Concept_5$.

The second case is if the concept extent's is empty (i.e. the concept which is not associated directly with a set of variant-differences), we randomly select only one variant-difference from each concept located immediately below and directly related to this concept. For example, considering the $Concept_6$ in Figure 3, we select randomly a variant-difference from $Concept_1$ and another one from $Concept_2$. Thus, for the $OnlineBank$ feature in $Concept_6$, its corresponding set of classes are in $Bank_V1.1$, $Bank_V2.0$ and $Bank_V1.2$. In both cases, differences among relevant variants are computed by lexically comparing their ECUs. For features of $Concept_5$, their corresponding classes is a set that contains all the *ECUs* of $V1.2$ that are not in $V1.1$.

3.3 Reducing the abstraction gap between feature and source code levels

In this step, we follow a two-step process to derive the *code-topics* from the common partition's classes and any minimal disjoint set of classes: computing similarity among classes and grouping similar classes into *code-topics* using FCA. In this paper, we use two kinds of source code information to compute the similarity: textual and structural information.

3.3.1 Computing textual and structural similarities

Textual similarity among given classes refers to textual matching between terms derived from identifiers related to these classes. We depend on VSM to compute the textual similarity. We follow VSM's steps described in section *2.2*. In VSM, each document represents a class. Each document is a list of all identifiers of its corresponding class. VSM computes the textual similarity between two class documents by using cosine similarity between their corresponding vectors. One of these documents has been treated as a query. Two documents are considered similar if the cosine of the angle of their corresponding vectors is greater than or equal to 0.70. This value represents the most widely used threshold for the cosine similarity [9]. After computing the cosine similarity among all class documents, we build a cosine similarity matrix whose columns and rows are identical and represent the class documents. An entry in this matrix refers to the cosine similarity value.

For structural similarity, we consider two classes are structurally similar if they have at least one of the following relations: inheritance, method call, shared method invocation and shared attribute access relations (all these relations are defined in section II.B). After computing the structural similarity among all classes, we build a structural similarity matrix whose columns and rows are identical and represent classes. An entry in this matrix refers to the structural similarity value. Each value is either 0 (there is no relation) or 1 (there is a relation).

3.3.2 Grouping similar classes into code-topics using FCA

FCA exploits the similarity among given classes to group them into *code-topics*. In this step, we define two formal contexts one of them for textual similarity and another one for combining textual and structural similarities. For textual similarity, the cosine similarity matrix was defined in the previous step represents a formal context where class documents represent objects and attributes at the same time. A relation between a document (as an object) and another document (as an attribute) refers to the cosine similarity value. Table 2 partially shows the formal context obtained by transforming the cosine similarity matrix for classes that implements the set of optional features located in the *Concept_5* of Figure 3. The cross sign refers to the similarity relation while null refers to no relation according to the threshold value. For combining textual and structural similarities, we define a formal context as above described however a relation between objects and attributes indicates to the textual or structural similarities. Table 3 partially shows a formal context generated by combing textual and structural similarities for the same classes considered in Table 2.

Table 2. A formal context for textual similarity.

	Bill_BillAcount	Bill_OldBills	Bill_PayPartially	Conversion_converter	Conversion_CurrencyInfo	Conversion_ConvertLimit	...
Bill_BillAcount	X		X				
Bill_OldBills		X					
Bill_PayPartially	X		X				
Conversion_converter				X	X		
Conversion_CurrencyInfo				X	X	X	
Conversion_ConvertLimit					X	X	
...							

Table 3. A formal context for combining textual and structural similarities.

	Bill_BillAcount	Bill_OldBills	Bill_PayPartially	Conversion_converter	Conversion_CurrencyInfo	Conversion_ConvertLimit	...
Bill_BillAcount	X	X	X				
Bill_OldBills	X	X	X				
Bill_PayPartially	X	X	X				
Conversion_converter				X	X	X	
Conversion_CurrencyInfo				X	X	X	
Conversion_ConvertLimit				X	X	X	
...							

Based on the formal contexts defined above, FCA is used as a clustering technique to group similar classes together via building a concept lattice. Each concept in the lattice contains a group of similar classes as extent. This group can be a candidate *code-topic*.

3.4 Mapping between features and code-topics

In this step, we separately apply LSI to establish traceability links between common feature partition and its possible corresponding *code-topics* as well as between each minimal disjoint set of optional features and its possible corresponding *code-topics*. LSI is applied by following the steps described in section 2.2 however we build LSI's corpus and queries as follows. LSI corpus consists of documents which each one corresponds to a code topic. Each document consists of the terms extracted from identifiers of classes that constitute a code-topic. For LSI's queries, we create for each feature a document which contains a feature name and description. Each feature document represent a query. LSI takes the corpus documents and queries as input. Then, LSI measures the similarity between the queries and documents using the cosine similarity. It returns a list of documents ordered by their cosine similarities values against each query. We consider again the same threshold value used in VSM for the cosine similarity.

3.5 Mapping between features and their classes

After establishing the traceability links between each feature and all its corresponding *code-topics*, we can easily link each feature with its implementing classes by decomposing each *code-topic* to its classes. For instance, if the feature *f1* is linked to two *code-topics*: *topic1*= {*c1, c2, c3*} and *topic2*= {*c1, c5, c6*}. By decomposing these topics into their classes; we can find that *f1* is implemented by five classes {*c1, c2, c3, c5, c6*}.

4 Experimental results and evaluation

4.1 Case studies

To validate our approach, we have applied it to seven variants of ArgoUML-SPL[1] , a large-scale system, and the first five releases of MobileMedia[2], a small-scale system. The ArgoUML-SPL is a Java open-source which supports all standard UML 1.4 diagrams. The ArgoUML-SPL's variants are generated from the same framework so that variants which share some features also share the same code. The selected variants support all ArgoUML's features. The ground truth links between features and their source code elements were determined by preprocessor directives to delimit the code associated to each feature. ArgoUML-SPL features are implemented at class level. The MobileMedia is a JAVA open source which manipulates multimedia on mobile devices. In our study, we have considered and analyzed variants 0 to 4 because they implement features at class level and vary in terms of the number of features available.

4.2 Performance of our approach

The most important parameter to LSI is the number of chosen term-topics. A term-topic is a collection of terms that co-occur frequently in the documents of the corpus. We need enough number of term-topics to capture real term relations. Too many term-topics lead to associate irrelevant terms and too few term-topics lead to loose relevant terms. In this work we cannot use a fixed number of term-topics for LSI, because we have different size of class sets. Thus, we use a factor K between 0.1 and 0.5 as well as between 0.01 and 0.05 to determine the number of term-topics. The number of term-topics (*#Term-topics*) is equal to $k \times D_{dim}$, where D_{dim} is a document dimensionality of the term-by-document matrix that is generated by LSI. We use different ranges for K because we have different sizes of variants and class sets.

[1] Available at: http://argouml-spl.tigris.org/
[2] Available at:http://www.ic.unicamp.br/ tizzei/mobilemedia/

Table 4. Average Precision, Recall and F-measure of FCT against CONV

Case Study	ArgoUML-SPL					
	Precision		Recall		F-measure	
K	FCT	CONV	FCT	CONV	FCT	CONV
0.01	52%	21%	95%	91%	67%	34%
0.02	52%	22%	91%	82%	66%	35%
0.03	50%	29%	87%	59%	64%	39%
0.04	51%	42%	81%	39%	63%	40%
0.05	57%	63%	76%	25%	65%	36%
Case Study	**MobileMedia**					
	Precision		**Recall**		**F-measure**	
K	**FCT**	**CONV**	**FCT**	**CONV**	**FCT**	**CONV**
0.1	71%	21%	100%	80%	83%	33%
0.2	71%	22%	99%	70%	83%	33%
0.3	81%	25%	85%	56%	83%	34%
0.4	81%	27%	81%	41%	81%	33%
0.5	86%	36%	77%	28%	81%	32%

Table 5. Results of Precision, Recall and F-measure of FCT against FL-PV.

Case Study	ArgoUML-SPL					
	Precision		Recall		F-measure	
K	FCT	FL-PV	FCT	FL-PV	FCT	FL-PV
0.1	71%	34%	51%	29%	59%	31%
0.2	60%	7%	6%	4%	11%	5%
0.3	89%	2%	3%	1%	6%	2%
0.4	60%	1%	2%	0%	4%	1%
0.5	80%	0%	1%	0%	2%	0%
Case Study	**MobileMedia**					
	Precision		**Recall**		**F-measure**	
K	**FCT**	**FL-PV**	**FCT**	**FL-PV**	**FCT**	**FL-PV**
0.1	85%	85%	100%	100%	92%	92%
0.2	85%	85%	100%	100%	92%	92%
0.3	93%	93%	93%	93%	93%	93%
0.4	93%	93%	93%	93%	93%	93%
0.5	96%	96%	89%	89%	93%	93%

Table 4 shows average precision, recall and F-measure results for our approach (Feature-to-Code traceability or FCT) and the conventional application of LSI (CONV) using different values of K for both case studies. Our approach results are obtained by combining textual and structural similarities to derive *code-topics*. As we can see, recall and precision results of FCT are better than those of CONV. This is attributed to two main reasons. First, FCT maps small sets of features to small sets of their respective source code classes in order to reduce the number of false positive links. Second, FCT bridges the abstraction gap between feature and source code levels using the code-topic. This leads to increase the number of retrieved links that are correct. The F-measure results confirm that FCT gives higher precision and recall compared with CONV in both case studies.

Table 5 summarizes results of FCT by combining textual and structural information to derive *code-topics* against the results of the most relevant work on the subject, called FL-PV [12]. This results represent the precision, recall and F-measure values for all variants' features of both case studies at different values of K. FL-PV considered reducing the LSI spaces as a factor to improve LSI results in a collection of software variants. Table 5 shows that FCT outperforms FL-PV in case of ArgoUML-SPL. This is attributed to the fact that FCT not only considers reducing LSI spaces like FL-PV but also reduces the abstraction gap between feature and source code levels. Also Table 5 shows that FCT and FL-PV give the same results in case of MobileMedia. This is because MobileMedia's features are implemented by a small number of non-cohesive classes and sometimes by only one class. Additionally, these classes don't have enough information to build *code-topics*.

We compare in Table 6 results of identifying *code-topics* by using only textual information and by combing textual and structural information. This results represent the average precision, recall and F-measure for locating features of both case studies for all K values in a range $[0.1, 0.5]$. We notice that combining textual and structural information gives better results than considering only textual information in case of ArgoUML-SPL. This means that by combining textual and structural information, we can acquire more relevant information to establish pertinent links between *code-topics* and features. However in case of MobileMedia both methods for identifying code-topics give the same results. This is due to that MobileMedia's features are implemented by a small number of non-cohesive classes.

Table 6. Results of textual information against combing textual and structural information for identifying code-topics

Case Study	Precision		Recall		F-measure	
	Combined	Textual	Combined	Textual	Combined	Textual
ArgoUML	72%	61%	13%	12%	22%	20%
MobileMedia	90%	90%	95%	95%	92%	92%

5 Related work

The works of Xue et al. [12], Ghanam et al.[13] Rubin et al.[14] belong to the second category. The most relevant work on the subject is proposed by Xue et al., called FL-PV. Their approach analyzes commonality and variability at feature and source code levels across software variants to reduce the LSI spaces. Then LSI is used to retrieve for each feature its corresponding possible source code units. Our approach differs from FL-PV by considering not only

reducing LSI spaces but also reducing the abstraction gap between feature and source code to retrieve more relevant source code elements for each feature. Ghanam et al. have put forward a method to keep the existing traceability links between feature model of a family of software products, produced by SPL engineering, and their source code up-to-date. Our proposed approach differs from Ghanam et al. where it starts from scratch and assumes no pre-existing links. Rubin et al. focused on only identifying code-feature traceability for distinguished features of two software variants implemented via code cloning and do not consider common features between them. In our previous work [15], we reduced LSI spaces by grouping all features and source code classes of a collection of software variants into only two partitions at feature and source code levels. The current work extends [15] by further reducing the LSI spaces into many sets of features and their sets of classes. Additionally, it reduces the abstraction gap between feature and source code level.

6 Conclusion and Future Work

In this paper, we presented a new approach to recover traceability links between features and object-oriented source code of a collection of software variants. The contribution of this paper is to improve the results of IR methods when they are applied to a collection of similar variants. additionally, we exploited together textual and structural information to retrieve more relevant links. The evaluation of our approach with a set of variants of two different systems showed that our approach outperforms the conventional application of LSI as well as the most recent and relevant work on the subject. The threat to the validity of our approach is that developers may not use the same vocabularies to name source code identifiers across software variants. This would mean that lexical matching at source code level would be affected. In the feature, we will employ this traceability links for impact analysis.

References

[1] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines." *Software Quality Journal*, vol. 17, no. 4, pp. 331–366, 2009.

[2] H. P. Breivold, S. Larsson, and R. Land, "Migrating industrial systems towards software product lines: Experiences and observations through case studies." in *EUROMICRO-SEAA*. IEEE, 2008, pp. 232–239.

[3] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," November 1990.

[4] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[5] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[6] M. Riebisch, "Towards a more precise definition of feature models," in *Modelling Variability for Object-Oriented Product Lines*, Norderstedt, 2003, pp. 64–76.

[7] A. Marcus and D. Poshyvanyk, "The conceptual cohesion of classes," 2005, pp. 133–142.

[8] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. USA: McGraw-Hill, Inc., 1986.

[9] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing." in *ICSE*, L. A. Clarke, L. Dillon, and W. F. Tichy, Eds. IEEE Computer Society, 2003, pp. 125–137.

[10] S. Apel and D. Beyer, "Feature cohesion in software product lines: an exploratory study," ser. ICSE '11. USA: ACM, 2011, pp. 421–430.

[11] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," ser. ICPC '07. USA: IEEE Computer Society, 2007, pp. 37–48.

[12] Y. Xue, Z. Xing, and S. Jarzabek, "Feature location in a collection of product variants." in *WCRE*. IEEE Computer Society, 2012, pp. 145–154.

[13] Y. Ghanam and F. Maurer, "Linking feature models to code artifacts using executable acceptance tests," ser. SPLC'10. Springer-Verlag, 2010, pp. 211–225.

[14] J. Rubin and M. Chechik, "Locating distinguishing features using diff sets," ser. ASE 2012. USA: ACM, 2012, pp. 242–245.

[15] H. Eyal-Salman, A.-D. Seriai, C. Dony, and R. Al-msie'deen, "Identifying traceability links between product variants and their features," ser. REVE'13, 2013, pp. 17–23.