Feature-to-CodeTraceability in a Collection of Product Variants Using Formal Concept Analysis and Information Retrieval

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony
UMR CNRS 5506, LIRMM
Université Montpellier 2 Sciences et Techniques
Place Eugène Bataillon, Montpellier, France
{Eyalsalman, Seriai, Dony}@lirmm.fr

Abstract— **Existing similar software variants, developed by ad-hoc reuse technique such as "clone-and-own", represent a starting point to build software product line (SPL) core assets. To re-engineer such legacy software variants into a SPL for systematic reuse, it is important to be able to identify a mapping between features and their implementing source code elements in different variants. Information retrieval (IR) methods have been used widely to support this mapping in single software product. This paper proposes a new approach to improve the performance of IR methods when they are applied on a collection of product variants. The novelty of our approach is twofold. In the one hand, it exploits what product variants have in common and how they differ to improve the accuracy of results given by IR methods. On the other hand, it reduces the abstraction gap between features and source code by introducing an intermediate level called "code topic" for increasing the number of correctly retrieved links. We have applied our approach on a collection of seven variants of a large-scale system by using the ArgoUML-SPL modeling tool. The experimental results showed that our approach outperforms the approaches that apply IR methods in conventional way as well as the most relevant work on the subject in the term of the most widely used metrics to evaluate IR methods: precision and recall.**

Keywords- Traceability; source code; features; product variants; latent semantic indexing; vector space model; formal concept analysis; product line.

I. INTRODUCTION

Software variants consist of similar software products that share some features, called common features, and also differ in others, called optional features. A feature is a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems [15]. Software variants often are developed by ad-hoc reuse techniques such as "clone-and-own" where an existing product is copied and later modified to meet incremental demands of customers. For instance, the Wingsoft Financial Management System (WFMS) was developed for Fudan University and then evolved many times to be adapted to different contexts. All variants of the WFMS systems have been used in over 100 universities in China [4].

At first glance, clone-and-own technique represents an easy and fast reuse mechanism. Indeed, it provides the ability to start from an already existing tested code and to make some modifications to produce a new variant. However, when the number of product variants and features grows, such ad-hoc reuse technique causes critical problems such as: maintaining each variant individually is complex as well as reusing features (i.e. sharing their implementations) from existing products into new product will be more complicated [7]. When these problems accumulate, it is needed to re-engineer product variants into a software product line (SPL) for systematic reuse. SPL is an engineering discipline supporting efficient development and maintenance of related software products [REF]. It manages common and optional features and promotes systematic software reuse from SPL's core assets (such as features, code, documentation and etc.). It capitalizes on the knowledge about available features, relationships among the features and traceability between the features and software artifacts that implement them [2].

In order to re-engineer a set of software variants into a SPL, it is important to be able to identify a mapping between features and their implementing source code elements (e.g., classes). Such a mapping is needed to understand product variants code and then automatically derive concrete products from SPL core assets by selecting features and consequently their corresponding source code. This mapping is known as traceability links recovery or feature location [12].

Information retrieval (IR) methods have been widely accepted to automate traceability recovery in single software product [7]. The conventional way for applying IR methods is to map all features of a software product to its entire source code. These features and this source code are called IR search spaces. In this paper, we propose an approach to improve the performance of IR-based methods when they are applied on a collection of software variants. The novelty of our approach is twofold. Firstly, it exploits commonalities and variabilities across product variants to reduce search spaces of IR. As a result, the accuracy of results given by IR methods increases by mapping less number of features to less implementation to prevent false positive links. Secondly, it reduces the abstraction gap between features and source code levels as a complementary part of reducing IR search spaces by introducing an intermediate level, called "code topic". Consequently, the number of correctly retrieved links increases because we map two similar software artifacts (features and

code topics).

The proposed approach uses lexical similarity and Formal Concept Analysis (FCA) to reduce IR search spaces by identifying common features and their associated source code elements, and grouping optional features and their associated source code elements into disjoint clusters. Regarding code topics, our approach again uses FCA with Vector Space Model (VSM) to derive code topics from the source code. Traceability links between a given cluster of features and the corresponding cluster of source code elements are recovered using Latent Semantic Indexing (LSI) taking into account derived code topics.

We have applied our approach on a collection of seven variants of a large-scale system by using the ArgoUML-SPL modeling tool. The experimental results showed that our approach outperforms the approaches that apply IR methods in conventional way as well as the most relevant work on the subject in the term of the most widely used metrics to evaluate IR methods: precision and recall.

The remainder of this paper is organized as follows. Section II presents background. Section III describes features versus object oriented building elements. Section IV presents the proposed approach. Section V represents experimental results and evaluation. Section VI discusses the threats to the validity of our approach. Section VII presents related work. Finally, section VIII presents conclusion and future work.

II. BACKGROUND

A. An Illustrative Example

As an illustrative example through this paper, we consider four text editor software variants as shown in the Table 1. *Editor_V1.0* supports just core features for any text editor: *Open*, *Create, Edit* and *Save* a file. *Editor_V1.1* has, in addition to the core features, *Search, Replace and Undo* features. *Editor_V1.2* supports not only core features but also new features (*Print, Help, Redo* and *Undo*). *Editor_V2.0* is an advanced text editor. It supports all previous features together.

B. Basics Concepts of FC A

Formal Concept Analysis (FCA) is a technique for data analysis and knowledge representation based on lattice theory. It identifies meaningful groups of objects that share common attributes as well as 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: extension and intension. The extension of a concept is the objects covered by the concept, while the intension comprises all the attributes, which are shared by all the objects covered by the concept [10].

In order to apply FCA, the formal context or incidence table of objects and their attributes is needed. The formal context is 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 $I \subseteq A$. $E = \{o \in O \mid \forall a \in I, (o, a) \in R\}$ is the extent of the concept. $I = \{a \in A \mid \forall o \in E, (o, a) \in R\}$ 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 concepts and concept lattices from a given formal context. In this work, we depend on Galois lattices that ignore empty concept [10]. Due to the space limitation, we will suffice the formal context and concept lattices displayed through the paper.

C. IR-based Traceability Recovery

Information retrieval (IR) methods have proved positive results to address traceability recovery [5]. The IR methods, such as Vector Space Model (VSM) and Latent Semantic Indexing (LSI), identify traceability links using the textual information from the software artifacts. For example, the keywords from features description may match keywords in the identifiers and comments of source code. One type of software artifact is treated as query and another type of artifact is treated as document. The IR methods rank these documents against queries by extracting information about the occurrences of terms within them. This information is used to find similarity between queries and documents [5]. In our case, the documents are source code classes and queries are features description where we create for each class and feature a document.

In VSM, documents and queries are represented by vectors of terms. Each term is a word appearing in the documents. The weight of a term can be computed with different schemes based on the task at hand. A collection of documents forms a term-by-document matrix with size $m \times n$, where m is the number of documents and n is the number of terms in all documents and queries. An entry $[i,j]^{th}$ indicates the association between the i^{th} term and j^{th} document. For a collection of queries, VSM also creates a term-by-query matrix with size $l \times n$, where l is the number of queries and n is number of terms in all documents and queries. The similarity between documents and queries is typically measured by the cosine of the angle between their corresponding vectors [5].

LSI extends VSM by using singular value decomposition technique (SVD). This technique is used to mitigate noise introduced by stop words like “the, an, above, etc.” and to overcome the two common issues of natural language processing i.e., synonymy and polysemy. SVD divides the term-by-document matrix to create LSI subspaces based on a parameter called “number of topics[1]”. For further details about SVD, the reader can refer to [13]. In the LSI space each document will have a corresponding vector. We use this vector representation to compute similarity. Like VSM, the textual similarity between documents and queries is measured by the cosine of the angle between their corresponding vectors [1].

Different strategies for identifying candidate traceability links are used, such as cut-points and thresholds. Our work uses a strategy based on similarity threshold values where all documents with a textual similarity value above or equal to the threshold are considered as candidate traceability links.

Promising results have been achieved using LSI to address concept location issue [19], and recovery of traceability links

[1] The term topic in LSI terminologies differs from the term topic in our approach.

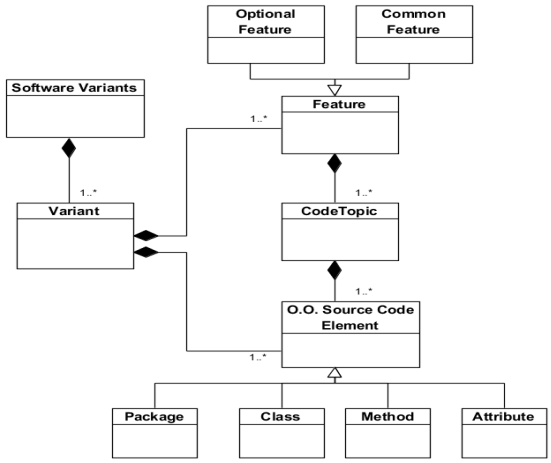


Fig.1. Feature-to-source code mapping model.

between source code and documentation [1]. In our approach, we use LSI to match feature descriptions with source code information.

III. FEATURE VERSUS OBJECT-ORIENTED ELEMENTS

For the purpose of this work, we adhere to the classification given by [14] which distinguishes three categories of features. Firstly, functional features express the behavior or the way users may interact with a product. Secondly, interface features express the product's conformance to a standard or a subsystem. Finally, parameter features express enumerable, listable environmental or non-functional properties. In our work, we focus on functional features.

As there are several ways to implement features (e.g., programming language level, meta language level) [6], we assume that functional features are implemented at the programming language level. Thus in an object-oriented source code, a functional feature can be implemented by packages, classes, methods, attributes, etc. As a class represents a main building unit in all object oriented languages, we assume that a functional feature is implemented at source code level by a set of classes.

Due to the abstraction gap between features and source code, recovering traceability links is a challenging problem. In order to overcome this problem, we propose an intermediate level, called "code topics". A code topic is a cluster of classes which are lexically similar and cover the same topic. A single code topic can represent a complete feature or some aspect of a feature. In addition, code topic can be shared between two or more features. The underlying intuition behind code topics is that a cluster of classes that implement a concept or a feature has a high probability to be linked lexically because domain knowledge represented by features is recorded in vocabularies used in identifiers and comments. By the time the traceability links between features and code topics were identified, we had also identified traceability links between features and source code where each code topic is a cluster of classes.

All concepts defined in our traceability recovery process are illustrated in feature-to-code mapping model of Fig.1.

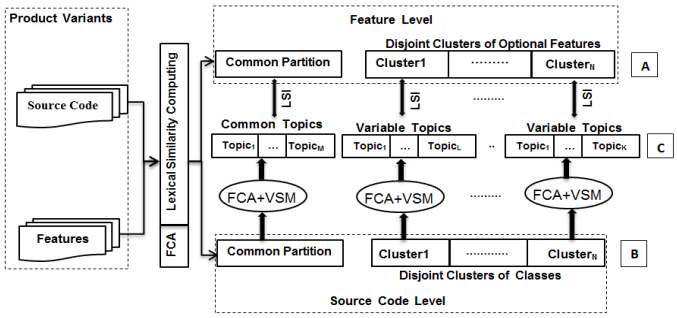


Fig.2. An overview of our approach.

IV. THE PROPOSED APPROACH

Fig.2 shows our traceability recovery process. This process takes as input source code and features of a set of given product variants. Each feature is identified by its name and description which consists of short paragraph. Such feature information is available in product variants due to the need for product customization. As an example of a feature description, *Edit* feature in our illustrative example can be described as follows: "*To allow user to do copy, paste and cut operations on a selected text in text area …etc*." .

In Fig. 2, FCA and Lexical similarity computing are used separately to divide product variants at feature and source code levels into a common partition and a set of disjoint clusters (See parts A and B in Fig. 2). At feature level, common features form common partition while optional features are organized as a set of disjoint clusters so that each cluster consists of one or more features. At source code level, a common partition is composed of classes associated to common features while each disjoint cluster is composed of classes associated to corresponding disjoint cluster of optional features. Also, FCA is used another time combined with VSM to derive code topics from source code. Common code topics in Fig.2 refer to topics derived from the common partition at source code level while variable code topics refer to topics derived from each disjoint cluster at source code level (see part C in Fig. 2). Traceability links between each cluster of features and its corresponding code topics are identified by LSI. After determining the topics that correspond to each feature, we easily determine classes that implement a feature by decomposing each topic to its classes. The following subsection explains each step on our recovery process in more details.

A. *Determining Common Partition at Feature and Source Code Levels*

As product variants share features and classes, our approach exploits this to identify common partition at feature and source code levels. At feature level, we rely on lexical similarity of features names and their descriptions for determining common partition at feature level of a collection of product variants. For a given set of features of product variants, we firstly define a subset of same name features. Secondly, as a feature may be renamed to respond to changes in software environment or to the adoption of different technology, we rely on the longest common subsequence (LCS) algorithm [16] to find features that have the same description but do not have the same name. We consider that two features identical if they have the same

subsequence terms of their description. In our illustrative example, all core features (e.g., *Open, Save, Edit and Create*) represent common partition at feature level.

At source code level, we analyze source code of a set of product variants itself in order to determine common partition at the source code level. The source code for each product variant is abstracted into a set of elementary construction units (*ECUs*). Each *ECU* has the following format:

ECU = PackageName_ClassName

This representation is inspired by the model construction operations proposed by [18]. Each product variant *Pi* is abstracted as a set of *ECUs*, i.e. $Pi=\{ECU_1,ECU_2,\ldots,ECU_n\}$. An *ECU* reveals any changes at package and class levels (e.g., add or remove packages or classes). These changes can reflect any variation at feature level (e.g. add or remove features).

In order to identify common *ECUs* shared by all product variants, we compare *ECUs* for all product variants together. This comparison process is done by conducting a lexical matching among *ECUs* for all variants such that *ECUs* for each product variant are lexically matched with *ECUs* for other variants. The shared *ECUs* across product variants represent common classes that represent the common partition at source code level.

B. *Grouping Optional Features and their Classes into Disjoint Clusters by FCA*

After determining common features and their associated classes, the reaming features and classes in each product variant represent optional features and their associated classes. The following steps reduce the search space related to these optional features and their classes using FCA.

1. *Identifing optional feature clusters Using FCA*

In order to reduce the search space related to optional features, we use FCA to group optional features via concept lattice into disjoint clusters. To achieve that, we define the formal context as follows: product scenarios (defined below) and optional features of a collection of product variants represent objects (extent) and attributes (intent) in the formal context respectively. A relation between a product scenario and an optional feature describes that the product scenario possess the optional feature. Table 1 shows a formal context of our illustrative example. In this context rows and columns are product scenarios and optional features respectively. Any entry in this context refers to that a scenario possesses certain optional features.

For any two product variants *P1* and *P2*, we build two product scenarios *P1-P2* (features exist only in *P1* but not in *P2*) and *P2-P1* (features exist only in *P2* but not in *P1*). Consider *Editor_V1.1 (V1.1)* and *Editor_V1.2 (V1.2)* as an example. Two product scenarios can be created as follows: *V1.1-V1.2 = {Search, Replace}* and *V1.2-V1.1 = {Print, Help, Redo}*. The product scenarios aim at identifying differences between each pair of products at feature level taking into account all combinations between product variants. Thus the formal context is constructed based on these differences and the concept lattice associates these differences with their product scenarios to form the lattice concepts.

TABLE 1. FORMAL CONTEXT FOR DESCRIBING TEXT EDITORS SCENARIOS.

	Search	**Replace**	**Undo**	**Redo**	**Help**	**Print**
V1.1-V1.2	x	x				
V1.2-V1.1				x	x	x
V1.1-V2.0						
V2.0-V1.1				x	x	x
…….						

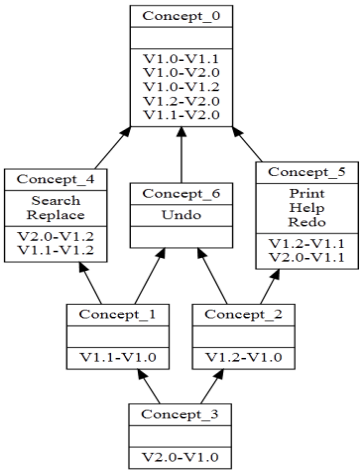


Fig.3. The concept lattice of formal context defined in Table 1.

Fig. 3 shows the concept lattice of formal context defined in Table 1. Each concept in the lattice consists of three fields. The upper field refers to concept name (generated automatically). The middle field represents disjoint cluster of optional features (intent). The bottom field shows scenarios (extent). Product scenarios help to determine which product variants must be compared to identify classes that implement a relevant cluster of optional features. Our approach compares two product variants by conducting a lexical matching between *ECUs* (classes) that abstracts these variants (see subsection IV.A).

We are interested in concepts associated with a set of features (such as the Concept_5 in Fig.3). They allow us to know how to obtain disjoint clusters of features and determine corresponding classes implementing these features.

2. *Identifying clusters of classes*

For a given cluster of optional features (i.e., a concept computed by FCA), we analyze the relevant scenarios to determine variants that have to compare. This allows us to isolate a cluster of classes that implements this given cluster of features by considering two cases. First, if a concept is directly associated with a set of scenarios (see *Concept_5* in Fig. 3); we randomly select only one scenario from this set. For instance, given a *Concept_5* (where the second scenario is chosen), our approach compares *Editor_V1.2* with *Editor_V1.1* to determine a cluster of classes that are present in *Editor_V1.2* but absent in *Editor_V1.1*. The resulting cluster of classes implements a cluster of features in the *Concept_5* (*Print, Help, Redo* features). The second case is if a concept does not associate directly with a set of scenarios (see *Concept_6* in Fig. 3), we randomly select only one scenario from each concept located immediately below and directly related to this concept. For

example, given a *Concept_6*, our approach selects a scenario from *Concept_1* and a scenario from *Concept_2*. For the *undo* feature in *Concept_6*, its corresponding classes are in both *Editor_V1.1* and *Editor_V1.2* but are not in *Editor_V1.0*. Our approach compares two product variants at source code level by conducting a lexical matching between ECUs that abstracts these variants.

C. Derivation of Code Topics by VSM and FCA

In the two previous steps, we considered just one factor to enhance LSI results (reducing LSI search space at feature and source code levels). This step considers a complement part of reducing LSI search space by reducing the abstraction gap between feature and source code levels using code topics.

Our approach groups classes of common partition and any disjoint cluster at source code level into a set of code topics. Our approach depends on a two-step process to derive code topics from source code: computing lexical similarity among classes using VSM and applying FCA. In the following subsections, we will explain each step in more details.

1. Lexical Similarity Computing Using VSM

As each code topic is a set of similar classes, a similarity measure is needed. In this paper, we consider lexical similarity as similarity measure.

Lexical similarity refers to textual matching between terms derived from identifiers and comments related to classes. In order to compute lexical similarity among classes, we create a document for each class. Each document contains lines of all identifiers and comments of corresponding class. These identifiers and comments should be manipulated such as tokenization, stop word removal and stemming performing.

We use VSM to measure lexical similarity between classes. VSM starts with a term-document matrix as mentioned earlier in section II. Each entry $a_{i,j}$ is the weight of term t_i in document d_j. In this paper, we used Term-Frequency/Inverse-Document-Frequency (TF/IDF) weight. The TF/IDF weight is often used in IR-based feature location approaches [13]. A geometric interpretation for term-document matrix is a set of document vectors as for each document there is a vector.

VSM computes lexical similarity between two documents (classes) one of them is a query using cosine similarity between their corresponding vectors. Two documents are considered similar, if the cosine of angle of their corresponding vectors greater is than or equal to 0.70. This value represents the most widely used threshold for cosine similarity [1]. After computing cosine similarity among all classes, we can build cosine similarity matrix which its columns and rows are identical and represent the documents. An entry in this matrix refers to cosine similarity value. This matrix is used an input to the next step.

2. Determining Code Topics Using FCA

The second use of FCA in our approach is to group similar classes into code topics. The formal context here is the cosine similarity matrix defined in the previous step: documents (classes) represent objects and attributes at the same time. A relation between a document (as object) and another document (as attribute) represent cosine similarity value. As FCA is a binary relationship, we use again the threshold (0.70) to transform the numerical values of the similarity matrix into binary formal contexts. This means that only pairs of documents having a similarity greater than or equal to 0.70 are considered similar. Table 2 shows a part of the formal context obtained by transforming the similarity matrix corresponding to *Concept_5* from Fig. 3. The cross sign refers to similarity relation while *null* refers to there is no relation according to the threshold value.

TABLE 2. A PART OF THE FORMAL CONCEPT OF CONCEPT_5.

	textEditor.print_print	textEditor.print_printSetting	textEditor.print_BufferPrinter Remove	textEditor.print_printer	textEditor.help_View	…….
textEditor.print_print	x	x	x	x		
textEditor.print_printSetting	x	x	x	x		
textEditor.print_BufferPrintRemove	x	x	x	x		
textEditor.print_printer	x	x	x	x		
textEditor.help_View					x	
……						

Concept_17
textEditor.print_print textEditor.print_printSetting textEditor.print_BufferPrinterRemove textEditor.print_printer
textEditor.print_print textEditor.print_printSetting textEditor.print_BufferPrinterRemove textEditor.print_printer

Fig4. An example of a code topic.

Concept_17 in Fig. 4 shows an example of a code topic. This concept is taken from the lattice corresponding to the formal context of *Concept_5* in Fig. 3. The extent of this concept represent a cluster of similar documents (classes) that are grouped together to form a code topic. Each line in this cluster represents an *ECU* (class). It is noticed that classes' names in *Concept_17* are similar and they also belong to the same package (*textEditor.print*). By manually analyzing these classes, we found that they represent the *print* feature in our illustrative example. The intent and extent of *Concept_17* are the same because objects and attributes in the formal context are identical (see Table 2).

D. Mapping features to code topics based on LSI

For a given set of features represented by common partition or any disjoint cluster at feature level, our approach used LSI to identify traceability links between these features and their associated code topics. Our applying of LSI is similar to [1]. It involves building LSI corpus and queries.

1. Building LSI Corpus

LSI corpus consists of documents which each one corresponds to a code topic. Each document consists of terms extracted from identifiers and comments of classes that represent the code topic. After building LSI corpus, LSI creates term-document matrix, where columns represent code topic documents and rows represent terms extracted from these

documents. Each term is weighted according to TD/IDF weight.

2. Building Queries

In our approach, LSI uses feature name and description as a query to retrieve code topics relevant to each feature. Our approach creates a document for each feature. Each document contains the feature name and description that must be normalized by splitting them into tokens, removing stop words and token stemming.

3. Establishing Traceability Links

LSI takes as input documents and queries generated in the two previous steps. It builds a vector of weights for each document (code topic) and query (feature). Each term is weighted by TF/IDF. Then, LSI measures the similarity between queries and documents using cosine similarity. It returns a list of documents ordered based on their cosine similarity against each query. We consider again the same threshold value used in VSM for cosine similarity, i.e., the retrieved documents have a cosine similarity with a query greater than or equal to 0.70.

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

V. EXPERIMENTAL RESULTS AND EVALUATION

In this section, we show the case study used for the evaluation of our approach, present the evaluation metrics and then discuss the experimental results.

A. Case Study and Experimental setting

To validate our approach, we applied it on a collection of seven variants of a large-scale system by using the ArgoUML-SPL modeling tool [8]. ArgoUML-SPL[2] is Java open-source which supports all standard UML 1.4 diagrams. It is well-documented and real implementation for each feature can be determined by preprocessor directives to delimit the code associated to each feature. The main advantage of ArgoUML-SPL is that it implements features at class level. The seven selected variants support all ArgoUML-SPL features. They consist of two common features (class diagram and cognitive support features) and six optional features (state, collaboration, use-case, activity, deployment and sequence diagram). Table 3 presents the size of selected product variants in terms of number of packages (NOP), number of classes (NOC) and number of lines of code (LOC).

TABLE 3. PRODUCT VARIANTS SIZE.

Products	*NOP*	*NOC*	*LOC*
Product#1	63	1455	99243
Product#2	67	1488	103969
Product#3	70	1554	107334
Product#4	74	1587	112060
Product#5	69	1541	110168
Product#6	72	1607	113533
Product#7	81	1666	118189
Product#8	65	1508	105442

[2] Available at http://argouml-spl.tigris.org/

B. Evaluation Measures

The effectiveness of IR methods is commonly measured by their precision, recall and F-measure. For a given query, precision is the percentage of correctly retrieved links to the total number of retrieved links. Recall is the percentage of correctly retrieved links to the total number of relevant links. F-measure makes a tradeoff 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 between [0, 1]. If recall 1, it means that all correct links are retrieved, though they could be retrieved links that are not correct. If precision 1, it means that all retrieved links are correct, though they could be correct links that are not retrieved. Higher precision, recall and F-measure mean better results [13].

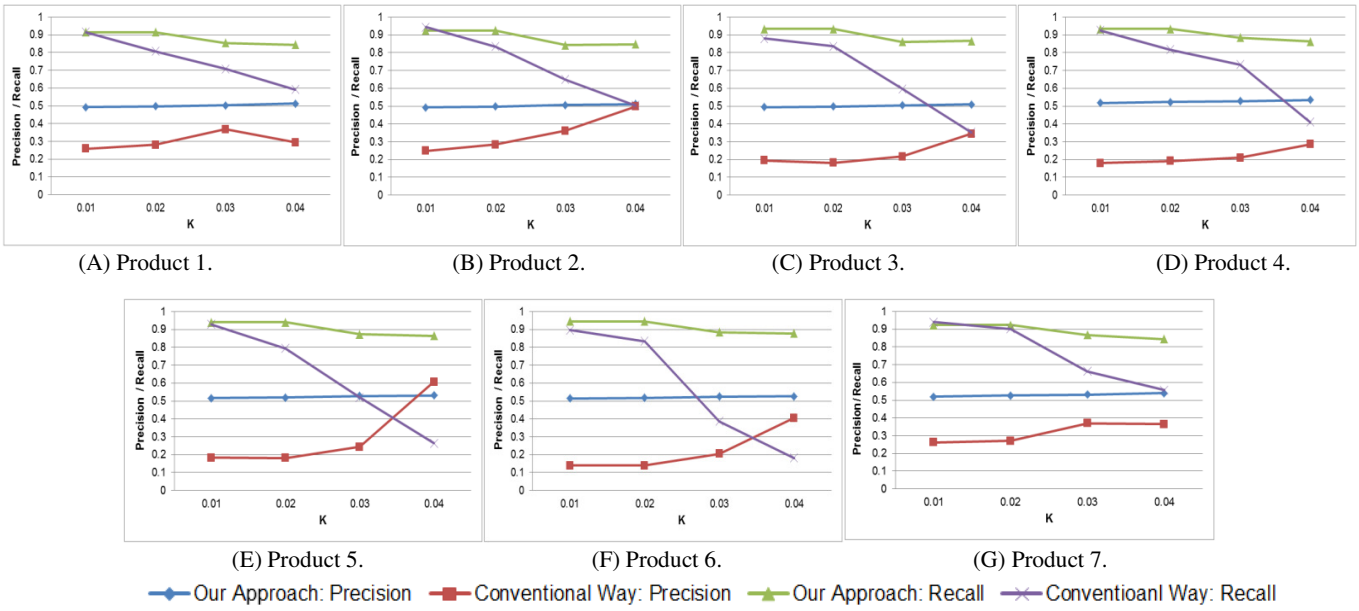
C. Performance of Our Approach

The most important parameter to LSI is the number of chosen topics. A topic is a collection of terms that co-occur frequently in the documents of the corpus. We need enough topics to capture real term relations. Too many topics lead to associate irrelevant terms and too few number of topics leads to lost relevant terms. According to Dumais et al. [19], the number of topics is between 235 and 250 for natural language. For a corpus of source code files, Poshyvanyk et al. [20] recommended topic's number of 750.

In this work we cannot use a fixed number of topics for LSI because we have different sizes of clusters. Thus, we use a factor k between zero and one to determine the number of topics. The number of topics (#topics) = $k \times Doc_d$, where Doc_d is document dimensionality of term-document matrix that is generated by LSI. We evaluate the performance of our approach for #topics where k= *0.1, 0.2, 0.3 and 0.4*.

A Fig.5 compares our approach against applying LSI in conventional way in term of the precision and recall metrics. The conventional way means applying LSI by considering each variant separately and also without taking into account the abstraction gap between features and source code levels. The graphs *A* to *G* given in Fig.5 correspond to *Product 1* to *Product 7* respectively. The X-axis in these graphs represents different values of k while Y-axis refers to precision and recall.

The graphs in Fig.5 indicate that our approach gives higher precision and recall than the conventional way. This is attributed to two reasons. Firstly, our approach maps less number of features to less implementation by reducing the LSI search space at feature and source code level. As a result, the accuracy of LSI results increases because of reducing the number of false positive links. Secondly, our approach bridges the abstraction gap between features and source code levels using code topics that can be a feature or some aspect of a feature. This means that our approach maps two similar software artifacts (features and code topics) which leads to increase the number of correctly retrieved links. By increasing the number of correctly retrieved links, the precision and recall also increase at the same time. Tables 4 and 5 show F-measure results for our approach and the conventional way respectively.



(A) Product 1. (B) Product 2. (C) Product 3. (D) Product 4.

(E) Product 5. (F) Product 6. (G) Product 7.

Fig.5. Comparing our approach with applying LSI in conventional way in term of the Precision and Recall metrics.

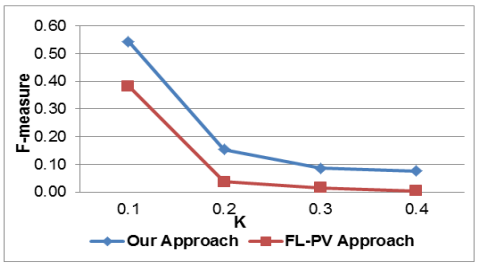


Fig.6. Comparing our approach with FL-PV in term of F-measure metric.

It is noticed that these results confirm that our approach gives higher precision and recall compared with the conventional way because when F-measure is high, this means that also precision and recall are high, too.

Fig.6 compares our approach with the most relevant work on the subject, called FL-PV [9], in term of F-measure metric. FL-PV considers reducing the LSI search at feature and source code levels as a parameter to improve LSI performance in a collection of product variants. Fig.6 shows that our approach outperforms FL-PL in term of F-measure. Since our approach gives higher F-measure values than FL-PV, this means that our approach also gives higher precision and recall than FL-PV according to the definition of F-measure. This is attributed to the fact that our approach not only considers reducing LSI search space like FL-PV but also reduces the abstraction gap between feature and source code level as a complementary part of reducing LSI search space. It can be seen that in Fig.8 when the number of topics (k) increases, results of our approach and FL-PV decreases. This is because of increasing the number of code topics results in capturing irrelevant terms.

Thanks to the integration of reducing LSI search space and bridging the gap between features and source code levels, our approach gives better results although ArgoUML-SPL features are similar and share code. For instance, there is a shared codes between *Activity* and *State* diagrams, between *Collaboration* and *Deployment* diagrams and *Cognitive Support* feature has crosscutting behavior across all other features.

VI. THREATS TO VALIDITY

The threat to the validity of our approach is that developers may not use the same vocabularies to name source code identifiers across product variants. This would mean that lexical matching at source code level would be affected. Nonetheless, when a company has to develop a new product that is similar, but not identical, to existing ones, an existing product is cloned and later modified according to new demands.

VII. RELATED WORKS

We classify feature location techniques into two categories. The first category is related to feature location in single software product. The second one includes techniques which support feature location in a collection of software

TABLE 4 F-MEASURE RESULTS OF OUR APPROACH.

K/Product	P1	P2	P3	P4	P5	P6	P7
0.01	0.64	0.64	0.65	0.67	0.67	0.67	0.67
0.02	0.64	0.65	0.65	0.67	0.67	0.67	0.67
0.03	0.63	0.63	0.64	0.66	0.66	0.66	0.66
0.04	0.64	0.64	0.64	0.66	0.66	0.66	0.66

TABLE 5.F-MEASURE RESULTS OF APPLYING LSI IN CONVENTIONAL WAY.

K/Product	P1	P2	P3	P4	P5	P6	P7
0.01	0.40	0.39	0.32	0.30	0.31	0.24	0.41
0.02	0.42	0.42	0.30	0.31	0.29	0.24	0.42
0.03	0.48	0.46	0.32	0.33	0.33	0.27	0.47
0.04	0.39	0.50	0.35	0.34	0.37	0.25	0.44

products. A comprehensive study about techniques in the first category can be found in [7] while the works of Ghanam et al. [3], Rubin et al. [11] and Xue et al. [9] belong to the second category.

Ghanam et al. have put forward a method to keep traceability links between features of a family of software products and their source codes up-to-date. They start from pre-existing links to make them up to date executable acceptance tests. Rubin et al focused on only locating distinguished features of two product variants implemented via code cloning and do not consider common features between them. In our recent work [17], we have proposed to consider not only two software variants but also consider all variants together. This allowed us to isolate both: the common and the optional features with their associated source code elements in each variant. LSI is used to map a group of features to its corresponding group of source code elements. As a result, the recall and precision of results given by LSI will be enhanced because of mapping less number of features to less implementation.

The most recent and relevant work on the subject is called FL-PV [9] (Xue et al work). FL-PV analyzes commonalities and differences at feature and source code levels across product variants to reduce search space related to features and respectively their source code elements into minimal disjoint partitions. Then LSI is used to retrieve code elements that implement a specific feature. The limit of FL-PV is if any partition consists of a large number of features, FL-PV remains inefficient because LSI search space also becomes large at feature and source code level.

Our approach differs from FL-PV by reducing the abstraction gap between features and source code levels as a complementary part of reducing the IR search spaces. Compared to Ghanam et al. work, the proposed approach starts from scratch and assumes no pre-existing links. With respect to Rubin et al.'s work, our approach not only locates distinguishing feature between two software variants but also locates all features across variants. Comparing with our recent work [17], the current work groups optional features and their associated source code elements across product variants into disjoint clusters at feature and source code levels.

VIII. CONCLUSION AND FUTURE WORK

We presented in this paper a new approach which combines FCA with IR, namely LSI and VSM, to establish traceability links between object oriented source code of a collection of product variants and given features of these variants. The contribution of this paper is twofold. Firstly, it improves the accuracy of results given by LSI via reducing the LSI search space at feature and source code levels. Secondly, it increases the number of correctly retrieved links by reducing the abstraction gap between feature and source code levels using code topics. The evaluation of our approach with seven variants of ArgoUML-SPL shows that our approach outperforms the approaches that apply IR methods in conventional way as well as the most relevant work on the subject (FL-PV).

In our future work, we plan to combine lexical similarity with structural similarity (e.g. method call and shared field access relationships) to enhance derived code topics. This requires a definition for structural similarity measure between source code elements.

REFERENCES

[1]. A. Marcus and J.I. Maletic. Recovering documentation-to-source code traceability links using Latent Semantic Indexing. ICSE 2003, pp.125-137.

[2]. P.Clements and L.Northrop. Software product lines: practices patterns. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2001.

[3]. Y. Ghanam and F.Maurer. Linking feature models to code artifacts using executable acceptance tests. In Proceedings of the 14th international conference on Software product lines: going beyond (SPLC'10), Jan Bosch and Jaejoon Lee (Eds.). Springer-Verlag, Berlin, Heidelberg,2010, 211-225.

[4]. P.Ye, X.Peng, Y.Xue and S. Jarzabek.: A Case Study of Variation Mechanism in an Industrial Product Line. ICSR. 2009,126-136.

[5]. D.Andrea, F.Fausto, O.Rocco and T.Genoveffa. Recovering traceability links in software artifact management systems using information retrieval methods. ACM Trans. Softw. Eng. Methodol. 16, 4,20007, Article 13 .

[6]. D.Beuche, H.Papajewski, S.Wolfgang. Variability management with feature models. *Sci. Comput. Program.* 53, 3 (December 2004), 333-352. 352.

[7]. B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey, JSME, 28 Nov, 2011.

[8]. M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in CSMR, T. Mens, Y. Kanellopoulos, and A. Winter, Eds. IEEE Computer Society, 2011, pp. 191–200.

[9]. X.Yinxing ,X. Zhenchang, J. Stan: Feature Location in a Collection of Product Variants. WCRE 2012: 145-154

[10]. G.Bernhard, W.Rudolf. *Formal Concept Analysis: Mathematical Foundations* (1st ed.). Springer-Verlag New York, Inc., Secaucus, NJ, USA.1997.

[11]. J. Rubin and M. Chechik. Locating distinguishing features using diff sets. In*Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*(ASE 2012). ACM, New York, NY, USA, 242-245.

[12]. G. Orlena and F. Anthony. An analysis of the requirements traceability problem. In Proceedings of 1st International Conference on Requirements Engineering (Colorado Springs, CO). IEEE Computer Society Press,1994, Los Alamitos, CA, 94–101.

[13]. S.Gerard and M.Michael. Introduction to Modern Information Retrieval. McGraw-Hill, Inc.,1996, New York, NY, USA.

[14]. M. Riebisch, "Towards a more precise definition of feature models," in Modelling Variability for Object-Oriented Prod- uct Lines, M. Riebisch, J. O. Coplien, and D. Streitferdt, Eds. Norderstedt: BookOnDemand Publ. Co, 2003, pp. 64–76.

[15]. K. Kyo, C.Cohen,H.James,N.William and P.Spencer.Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990,Carnegie Mellon University.

[16]. L. Bergroth and H. Hakonen and T. Raita. A Survey of Longest Common Subsequence Algorithms. SPIRE 2000, pp. 39–48.

[17]. E.Hamzeh, S.Abdelhak-Djamal, D.Christophe and A.Ra'Fat. Identifying Traceability Links between Product Variants and Their Features . CSMR Workshop(17th European Conference on Software Maintenance and Reengineering) March 5–8, 2013, Genova, Italy.

[18]. X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting model inconsistency through operation-based model construction," in ICSE, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds.ACM, 2008, pp. 511–520.

[19]. S.T. Dumais. LSI meets TREC: A status report, in Proceeding of Text Retrieval Conference, pp. 137-152. 1992.

[20]. D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Guéhéneuc, and G. Antoniol. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. ICPC, pp. 137-148, 2006.