Toward Reengineering Legacy Software Variants into Software Product Line: Feature-to-Code Traceability

Hamzeh Eyal-Salman

UMR CNRS 5506, LIRMM, University of Montpellier 2 for Sciences and Technology, Place Eugne Bataillon, Montpellier, France

Abstract. Existing software product variants, developed by ad-hoc reuse techniques such as “clone-and-own”, represent a starting point to build a software product line (SPL) core assets for systematic reuse. In order to re-engineer such legacy systems into a SPL, it is important to identify a mapping between features and their implementing source code elements in the different variants. Information retrieval (IR) methods have been widely used to support this mapping in single software. We propose a new approach to improve the performance of IR methods when they are applied to a collection of product variants. The novelty of our approach is twofold. On the one hand, it exploits what product variants have in common and how they differ to improve the accuracy of results given by the 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. To validate our approach, we have applied it on a set of variants for a large-scale system, ArgoUML-SPL. The experimental results showed that our approach outperforms the conventional application of IR techniques as well as the most relevant work on the subject.

Keywords: Traceability, source code, object-oriented, features, product variants, latent semantic indexing, vector space model, formal concept analysis, product line.

1 Introduction

Software product variants are a set of similar software products which share some features, called common features, and differ in other ones, called optional features, to accommodate specific demands of customers in a particular domain[1]. A feature is a user-visible characteristic of software systems[2]. Features are used to describe the commonalities and variabilities of products variants. For example, GPS systems share common features (e.g., determining directions and positions) but they differ in other features (e.g., live traffic information and shortest time routing).

Usually software variants are developed by ad-hoc reuse techniques such as “clone-and-own” where an existing product is copied and later modified independently from the original version to develop a new product[1]. Such reuse technique provides the ability to start from already existing tested code and make necessary modification to it. However, this technique causes critical maintenance problems such as maintaining each variant individually is time consuming as well as reusing features (sharing their implementations) from existing products into a new product will be more complicated[3]. To overcome these problems, it is important to re-engineer such legacy software variants into a software product line (SPL) for systematic reuse. SPL engineering is an engineering discipline supporting efficient development and maintenance of related software products. It manages common and optional features and promotes systematic software reuse from SPLs core assets (such as features, code, documentation and etc.). It exploits knowledge about features, relationships among the features and traceability between the features and software artifacts that implement them to develop short time-to-market products [4].

To re-engineer software variants into a SPL, we have to identify a mapping between features and their implementing source code elements (e.g. classes)[4]. This mapping is needed to understand software variants code and then automatically derive concrete products from SPL core assets. This mapping is called traceability links recovery or feature location. In this paper, we use the terms "traceability links recovery" and "feature location" interchangeably.

Information retrieval (IR) has been long recognized as a method to automate traceability recovery. The IR-based methods, such as latent semantic indexing (LSI), rely on the analysis of textual information derived from software artifacts to identify traceability links[5]. For example, the keywords from feature descriptions may match keywords in the identifiers and comments of source code. Indeed, all IR methods recover the traceability links in single software. The conventional application of IR methods is to match all features (i.e., their descriptions) of a software product to its entire source code. These features and the associated source code are called IR spaces.

In this paper, we propose a new approach to improve the performance of IR methods when they are applied to identify traceability links between features and object-oriented source code in a collection of software variants. The novelty of our approach is twofold. Firstly, it exploits commonality and variability of software variants at feature and source code levels to increase the accuracy of IR results by reducing IR spaces. When software variants are considered together, we can determine common features (resp. their associated source code elements) and group optional features (resp. their associated source code elements) into disjoint clusters. By considering a set of product variants together, we can discriminate relevant and non-relevant source code elements for feature(s). This discrimination improves the accuracy of IR results by reducing false positive links via mapping small sets of features to small sets of source code elements. Secondly, it reduces the abstraction gap between feature and source code levels to increase the number of retrieved links that are correct by introducing an intermediate level, called "code-topic" . A "code-topic" is a cluster of classes that are grouped based on their contents (i.e., the terms in their identifiers and comments) to implement a functionality or functionalities. Consequently, a feature (i.e., its description) can be matched to a set of code-topics (i.e., their source code information)where we assume that a feature can correspond to one or more functionalities that are related. 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. By grouping a set of classes that cover the same topic, we can get enough information (i.e., classes' comments and identifiers) to match together these classes to a feature description where a feature implementation spans multiple classes.

The proposed approach effectively combines Formal Concept Analysis (FCA); IR techniques namely Vector Space Model (VSM) and Latent Semantic Indexing (LSI); and lexical similarity computing. FCA and lexical similarity are used separately to reduce IR spaces into disjoint clusters of features and their corresponding cluster of source code. Again FCA is used together with VSM to derive "code-topics" from each disjoint cluster of source code. Traceability links between features and *code-topics* are recovered using LSI. As a result, we can easily determine classes relevant to each feature by decomposing each *code-topic* to its classes where we assume that the feature is implemented by a set of classes. We call our proposal as Feature-to-Code Traceability or FCT for short.

The remainder of this paper is organized as follows: background and related work are presented in section 2. Section 3 gives an overview about our solution. Section 4 presents experimental results and evaluation. Finally, section 5 concludes our work.

2 Background and Related Work

This section is devoted to give an overview about textual analysis and Formal Concept Analysis (FCA). It also discusses related works.

2.1 Textual Analysis

Textual analysis refers to textual similarity among documents. In our case, the documents are both source code classes and feature descriptions where we create for each class and feature a document. The textual similarity is computed using the occurrence of terms in the documents. Two documents are considered similar if they share a large number of terms. Different IR methods have been proposed in the context of program comprehension such as LSI and VSM. However, they share four main steps[6]: creating a corpus, preprocessing, indexing, querying. Firstly, the 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 partition of source code such as a method, class or package. Secondly, the corpus undergoes a preprocessing step. The preprocessing involves normalizing the documents content such as stop word removal and stemming performing. In the third step, a term-by-document matrix is created by using the corpus. The matrix's columns correspond to the documents in the corpus and the rows represent terms that are extracted from these documents. The values in the matrix indicate the number of occurrences the term in the document. Finally, a user makes a query that describe the feature to be located. Each query is manipulated by the same preprocessing techniques as the software corpus.

Promising results have been achieved using LSI to address concept location issue [7], and recovery of traceability links between source code and documentation[3]. 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. Based on this strategy, all documents with a cosine similarity value above or equal to the threshold are considered as candidate traceability links.

2.2 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 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 [8]. In our case, we use FCA to group optional features and their associated source code elements into disjoint clusters (i.e. concepts).

2.3 Related Works

We classify feature location techniques into two categories. The first category support feature location in single software product. The second one includes techniques which support feature location in a collection of software products. A comprehensive study about techniques in the first category can be found in[3] while the works of Ghanam et al.[8], Rubin et al.[9], Ziadi et al.[10] and Xue et al. [11] belong to the second category.

Ghanam et al. [8] have put forward a method to keep traceability links between features of a family of software products and their source codes up-to-date. When SPL evolves, traceability links become broken or outdated due to evolution of features and source code. Their method is based on executable acceptance tests (EAT). EAT refers to English-like specifications (such as: scenario and story tests). These EATs represent the specifications of a given feature and can be executed against the system to test the correctness of its behavior. Their approach starts from pre-existing links to make them up to date while our approach differs from this work as we start from scratch and assume no pre-existing links.

Rubin et al. [9] focused on only locating distinguished features of two software variants implemented via code cloning. Distinguished features refer to the features present in one variant and absent in another. Their work aims at identifying code regions that only implement the distinguished features to improve the accuracy of existing feature location techniques. These regions are called `diff` set and computed by comparing the code of a variant that contains these features to the one that does not. Their approach does not consider common features between two variants. Therefore, the approach is not much useful to enhance the accuracy of existing feature location techniques when these techniques are applied to locate common features.

Ziadi et al. [10] proposed an approach to automate feature identification from the source code of similar products variants. Their approach aims at locating a region of code that can implement a feature without take into account feature information such as name and description. The main limitation of their work is that it identifies common features as a single feature called base feature.

The most relevant work on the subject is proposed by Xue et al. [11], called FL-PV approach. FL-PV 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 elements.

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. Regarding Ghanam et al.'s work where the proposed approach starts from scratch and assumes no pre-existing traceability links. With respect to Rubin et al.'s work, our approach not only locates distinguishing features between two software variants but also locates common features across variants. Our approach differs from Ziadi et al.'s work by exploiting features information (i.e., name and description) that is usually available in software variants due to the customization process to locate not only optional features but also common features in the source code. By exploiting this information about features, we can establish pertinent links between features and source code elements.

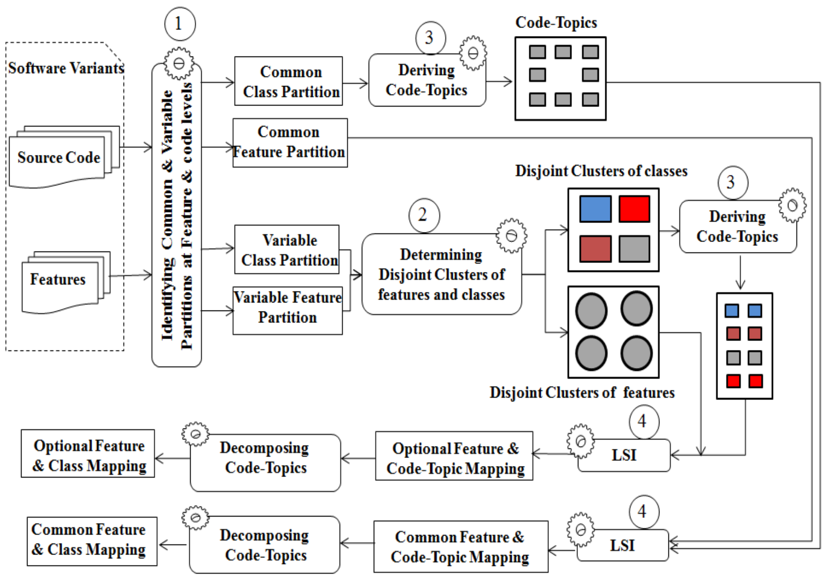
3 The proposed approach

In this section, we present the basic assumptions of our work and describe our solution to recover the traceability links.

3.1 Basic assumptions

For the purpose of this work, we adhere to the classification given by[12] which distinguishes three categories of features: functional, interface and parameter features. Functional features express the behavior or the way users may interact with a product. Interface features express the product's conformance to a standard or a subsystem. Parameter features express enumerable, listable environmental or non-functional properties.

We also restrict ourselves to object-oriented systems. Thus in an object-oriented source code, a feature either common or optional can be implemented by using packages, classes, methods, attributes, etc. As the class represents the main building unit in all object-oriented languages and most often developers



Software Variants
Source Code
Features
1
Identifying Common & Variable Partitions at Feature & code levels
Common Class Partition
Common Feature Partition
Variable Class Partition
Variable Feature Partition
3
Deriving Code-Topics
Code-Topics
2
Determining Disjoint Clusters of features and classes
Disjoint Clusters of classes
Disjoint Clusters of features
3
Deriving Code-Topics
4
LSI
Optional Feature & Code-Topic Mapping
Decomposing Code-Topics
Optional Feature & Class Mapping
4
LSI
Common Feature & Code-Topic Mapping
Decomposing Code-Topics
Common Feature & Class Mapping

Fig. 1. An overview of our approach.

think about the class as a set of responsibilities that simulate a concept or functionality from the application domain; we assume that the functional feature is implemented by a set of classes.

3.2 Feature-To-Code Traceability Process: Overview

Figure 1 shows our traceability recovery process. This process takes two inputs: object-oriented source code and feature description of a set of given product variants. Each feature is identified by its name and the description which consists of short paragraph. Such feature information is available in product variants due to the need for product customization.

Our traceability recovery process consists of four main steps. The first two steps aim to reduce LSI spaces. Firstly, lexical similarity computing separates all features and classes of a given set of software variants into common and variable partitions (see step 1 in Figure 1). At feature level, common and variable partitions consist of common and optional features respectively. At source code level, common and variable partitions consist of classes that implement common and optional features respectively. Next, the variable partitions at feature and source code levels are fragmented into disjoint clusters using FCA (see step 2 in Figure 1). In the third step, FCA and VSM are combined together to derive “code-topics” from each disjoint cluster of classes computed that are computed in the previous step (see step 3 in Figure 1). Finally, the traceability links between features and their possible corresponding “code-topics” are established using LSI. These links are used as a means to connect features with their source code classes.

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.

4 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.

4.1 Case study and experimental setting

To validate our approach, we have applied it to a seven variants of a large-scale system, called ArgoUML-SPL[1]. The ArgoUML-SPL is a JAVA open-source modeling tool which supports all standard UML 1.4 diagrams. It is well-documented and real implementation for each feature can be determined by the preprocessor directives to delimit the code associated with each feature. This pre-compilation process allows us to establish the truth links (real implementation for each feature) for evaluating the effectiveness of our approach. The main advantage of ArgoUML-SPL is that it implements features at the 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 1 presents the size of selected product variants in terms of the number of packages (NOP), the number of classes (NOC) and the number of lines of code (LOC).

Table 1. Source code characteristics for each variant.

ArgoUML-SPL Variants			
Variants	**NOP**	**NOC**	**LOC**
Variant#1	67	1,488	103,969
Variant#2	70	1,554	107,334
Variant#3	74	1,587	112,060
Variant#4	69	1,541	110,168
Variant#5	72	1,607	113,533
Variant#6	81	1,666	118,189
Variant#7	65	1,508	105,442

To analyze the source code of given variants, we used Eclipse JDT/DOM APIs to build abstract syntax tree (AST) that can be queried to extract source code information. For LSI[2] and VSM[2], I have implemented them according to the

[1] http://argouml-spl.tigris.org/

[2] Available at https://code.google.com/p/lsi-vsmodel/

[2] Available at https://code.google.com/p/lsi-vsmodel/

description in section II. For FCA, we depends on Galois sub-hierichal (GSH[3]) tool.

4.2 Evaluation metrics

The effectiveness of IR techniques is commonly measured by their precision, recall and F-measure [6]. For a given query, precision measures the accuracy of IR results. Precision is the percentage of correctly retrieved links to the total number of retrieved links. Recall quantifies the number of correctly retrieved links that are correct. Recall is the percentage of correctly retrieved links to the total number of relevant links. The 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 metrics have values in the range [0, 1]. If precision is equal to 1, it means that all retrieved links are correct, though they could be correct links that are not retrieved. If recall is equal to 1, it means that all correct links are retrieved, though they could be retrieved links that are not correct. Higher precision, recall and F-measure mean better results.

4.3 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 leads to loose relevant terms. According to Dumais et al. [13] , the number of term-topics is between 235 and 250 for natural language. For a corpus of source code files, Poshyvanyk et al. [14] recommended that the number of term-topics is 750. In this work we cannot use a fixed number of term-topics for LSI, because we have different size of feature and class clusters. Thus, we use a factor K between 0.1 and 0.4 as well as between 0.01 and 0.04 to determine the number of term-topics. The number of term-topics ($\#Topics$) is equals to $K \times doc_d$, where doc_d is document dimensionality of term-by-document matrix that is generated by LSI.

Table 2 summarizes the average precision, recall and F-measure values for all variants at different values of K for our approach (FCT) and the conventional application of LSI (Conv.). As we can see, recall and precision results of FCT are better than those of the conventional application of LSI. This is attributed to two main reasons. Firstly, FCT maps small sets of features to small sets of their respective source code classes by reducing the LSI spaces at feature and source code levels. As a result, the accuracy (precision) of retrieved links increases because of reducing the number of false positive links. Secondly, FCT bridges the abstraction gap between feature and source code levels using the *code-topic* that can be a feature or some aspect of a feature. This means that FCT maps two similar software artifacts (features and *code-topics*), which leads to increase the

[3] Available at https://code.google.com/p/erca/

Table 2. Average Precision, Recall and F-measure of FCT against the conventional application of LSI.

Case Study	**ArgoUML-SPL**					
	Precision		**Recall**		**F-measure**	
K	**FCT**	**Conv.**	**FCT**	**Conv.**	**FCT**	**Conv.**
0.01	51%	22%	93%	80%	66%	46%
0.02	51%	22%	93%	71%	66%	46%
0.03	52%	19%	87%	55%	65%	46%
0.04	52%	0%	86%	38%	65%	36%
0.05	53%	0%	73%	25%	61%	33%

Table 3. Average 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	75%	64%	43%	27%	54%	38%
0.2	63%	40%	9%	2%	15%	4%
0.3	57%	44%	5%	1%	9%	2%
0.4	62%	50%	4%	0%	8%	0%
0.5	67%	0%	0%	0%	1%	0%

number of retrieved links that are correct. The F-measure results confirm that FCT gives higher precision and recall compared with the conventional application of LSI. This is because when F-measure is high, then precision and recall are also high (according to the F-measure definition).

In Table 3, we compare results of FCT and the most relevant work on the subject, called FL-PV [11]. This results represent the average precision, recall and F-measure values for all variants at different values of K. FL-PV considers reducing the LSI spaces at feature and source code levels as a factor to improve LSI results in a collection of software variants. Table 3 shows that FCT outperforms FL-PV. 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. This means that reducing LSI space is not the only important factor to improve LSI results but also reducing the abstraction gap is another important factor.

5 Conclusions

in this paper, we had presented a new approach which combines FCA with IR methods 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 work is twofold. Firstly, it explores what software variants have in common and how they differ to enhance the accuracy of IR results. Then, it reduces the abstraction gap between feature and source code levels using "code-topics" in order to increase the number of retrieved links that are relevant. Our evaluation on a set of variants of a large-scale system, ArgoUML-SPL modeling tool, showed that our approach outperforms the conventional application of IR techniques as well as the most relevant work on the subject.

In the future, we plan to combine textual similarity with structural similarity (e.g. method call and shared field access relationships) to enhance derived *code-topics*.

References

1. Ye, P., Peng, X., Xue, Y., Jarzabek, S.: A case study of variation mechanism in an industrial product line. In: Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering. ICSR '09, Berlin, Heidelberg, Springer-Verlag (2009) 126–136
2. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. (November 1990)
3. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Evolution and Process **25**(1) (2013) 53–95
4. Clements, P.C., Northrop, L.M.: Software product lines: practices and patterns. Addison-Wesley (2001)
5. Lucia, A.D., Fasano, F., Oliveto, R., Tortora, G.: Recovering traceability links in software artifact management systems using information retrieval methods. ACM Trans. Softw. Eng. Methodol. **16**(4) (September 2007)
6. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw-Hill, Inc., New York, NY, USA (1986)
7. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In Clarke, L.A., Dillon, L., Tichy, W.F., eds.: ICSE, IEEE Computer Society (2003) 125–137
8. Ghanam, Y., Maurer, F.: 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, Berlin, Heidelberg, Springer-Verlag (2010) 211–225
9. Rubin, J., Chechik, M.: Locating distinguishing features using diff sets. ASE 2012, New York, USA, ACM (2012) 242–245
10. Ziadi, T., Frias, L., da Silva, M.A.A., Ziane, M.: Feature identification from the source code of product variants. CSMR '15, USA, IEEE (2012) 417–422
11. Xue, Y., Xing, Z., Jarzabek, S.: Feature location in a collection of product variants. In: WCRE. (2012) 145–154
12. Riebisch, M.: Towards a more precise definition of feature models. In Riebisch, M., Coplien, J.O., Streitferdt, D., eds.: Modelling Variability for Object-Oriented Product Lines. BookOnDemand Publ. Co, Norderstedt (2003) 64–76
13. Dumais, S.T.: Lsi meets trec: A status report. In: REC1. (1993) 137–152
14. Poshyvanyk, D., gal Guhneuc, Y., Marcus, A., Antoniol, G., Rajlich, V.: Combining probabilistic ranking and latent semantic indexing for feature identification. In: ICPC'06. (2006) 137–148