Feature Location in a Collection of Product Variants: Combining Information Retrieval and Hierarchical Clustering

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony
UMR CNRS 5506, LIRMM, University of Montpellier 2 for Sciences and Technology, France
E-mail: {Eyalsalman, Seriai, Dony}@lirmm.fr

Abstract

Locating source code elements relevant to a given feature is an important step in the process of re-engineering software variants, developed by an ad-hoc reuse technique, into a Software Product Line (SPL) for systematic reuse. Existing works on using Information Retrieval (IR) techniques do not consider the abstraction gap between feature and source code levels. In our recent work, we have improved the effectiveness of IR-based feature location by introducing an intermediate level between feature and source code levels, called "code-topics". We used Formal Concept Analysis (FCA) to identify such "code-topics". In this paper, we investigate the results of using Agglomerative Hierarchical Clustering (AHC) algorithm to identify code-topics. In our experimental evaluation, we show that AHC significantly increases the recall of feature location with a minor decrease of precision compared to FCA.

Keywords: *Software product line, product variants, feature location, FCA, AHC, IR.*

1 Introduction

Software product variants are a set of similar software products developed by ad-hoc copying with adaptations to meet new demands of customers [1]. They share some features, *called mandatory features*, and also support different, customer-specific features, called *optional features*. A feature is "a prominent or distinctive user-visible aspect, quality or characteristic of a software system" [2]. As numbers of variants and features grow, maintaining such software variants and developing new variants becomes more difficult and expensive over time. Therefore, a transition to a systematic reuse approach, such as Software Product Line Engineering (SPLE) becomes necessary.

SPLE is a engineering discipline providing methods to promote systematic software reuse for developing short time-to-market and quality products in a cost-efficient way [3]. These products are known as SPL. SPL is rarely developed from scratch. It is often built by exploiting artifacts of pre-existing software product variants. To re-engineer software variants into a SPL, it is important to locate source code elements (e.g., classes) that implement each feature [4]. Feature location is needed to understand the source code of software variants and support product derivation from SPL core assets[3]. Manually locating feature implementations is an error-prone and time consuming task. This is because maintainers must understand several software artifacts to decide which feature is implemented by which source code elements.

Information Retrieval (IR) techniques have been widely used for feature location [5][6]. In our recent work [4], we have improved the effectiveness of IR-based feature location in a collection of product variants 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 reveal source code intention. A *code-topic* can be a functionality implemented by the source code and provided by a feature. In our recent work [4], we have combined a technique called Formal Concept Analysis (FCA) and IR to identify such *code-topics*. In this paper, we propose to investigate the results of using Agglomerative Hierarchical Clustering (AHC) to identify "code-topics" .

The rest of this paper is organized as follows. Section 2 presents necessary background to understand our proposal. Then, Section 3 shows how AHC can be used to identify *code-topics*. In sections 4 and 5, we present experimental evaluation and discuss related work, respectively. Finally, Section 6 concludes the paper.

2 Background

2.1 IR-based Feature Location

IR-based feature location methods exploit source code information (identifier and comments) to locate a feature's implementation. These methods works by conducting lexical matching between source code information and feature information (i.e., feature description). Different IR techniques, such as Vector Space Model (VSM) and Latent Semantic Indexing (LSI), have been proposed in the context of locating features in the source code. These techniques share four steps: *corpus creation, preprocessing, indexing* and *querying*. For more details, the reader can refers to [7]. In both LSI and VSM, the textual similarity between source code documents (corpus) and features documents (queries) is measured by the cosine of the angle between their corresponding vectors. Both LSI and VSM return a ranked list of source code documents against each feature document.

The effectiveness of IR techniques is measured by their *recall, precision* and *F-Measure* [7]. For a given query, recall is the percentage of retrieved documents that are relevant to the total number of relevant documents, while precision is the percentage of retrieved documents that are relevant to the total number of retrieved documents. F-Measure defines a tradeoff between precision and recall so that it gives a high value only in case where both recall and precision are high. All measures have values in [0,1].

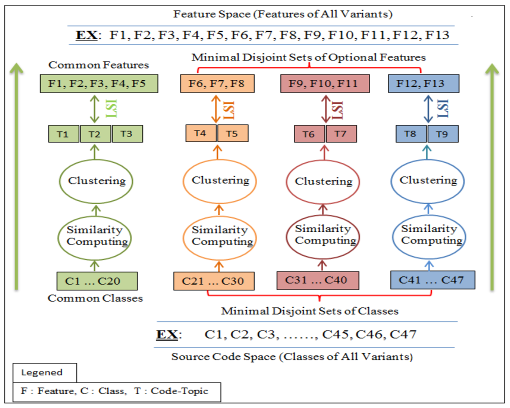
2.2 IR-based Feature Location in a Collection of Product Variants: An Overview

In this section, we give an overview of our recent work [4], in order to more easily understand our proposal.

2.2.1 Basic Assumptions

We focus on functional features that express the behavior or the way users may interact with a product. We restrict ourselves to object-oriented systems. A functional feature can be implemented by a set of packages, classes, methods and attributes. A *class* represents a main building unit in all object-oriented programming where it encapsulates functionality and data. Moreover, developers typically think of a *class* as a set of responsibilities that simulate a concept or functionality from the application domain. Consequently, we assume that a functional feature is implemented by a set of classes.

The functional feature's implementation spans multiple classes. We assumed that classes that contribute to implement a feature have shared terms and called near to each other. By grouping these classes into a cohesive unit based on their natural language content, we can get more relevant information describing features implemented by these classes. Therefore, we proposed the *code-topic*, as a coherent cluster of similar classes that are grouped based on their textual contents to implement a functionality. The *code-topic* constitutes an intermediate level that bridges the abstraction gap between feature and source code levels. Consequently, the functional feature (i.e., its description) can be textually matched to a set of *code-topics* (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 individually.



Feature Space (Features of All Variants)
EX: F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13
Minimal Disjoint Sets of Optional Features
Common Features
F1, F2, F3, F4, F5
F6, F7, F8
F9, F10, F11
F12, F13
LSI
LSI
LSI
LSI
T1 T2 T3
T4 T5
T6 T7
T8 T9
Clustering
Clustering
Clustering
Clustering
Similarity Computing
Similarity Computing
Similarity Computing
Similarity Computing
C1 ... C20
C21 ... C30
C31 ... C40
C41 ... C47
Common Classes
Minimal Disjoint Sets of Classes
EX: C1, C2, C3, ..., C45, C46, C47
Source Code Space (Classes of All Variants)
Legened
F : Feature, C : Class, T : Code-Topic

Figure 1. An Overview of the Code-Topic Identification Process.

2.2.2 Code-Topics Identification

In our recent work [4], we follow to strategy to improve IR-based feature location. Firstly, we determined mandatory features (resp. their classes) across software variants and grouped optional features (resp. their classes) into minimal disjoint sets at feature and source code levels. Secondly, we reduced the abstraction gap between feature and source code levels by introducing the *code-topic* as an intermediate level. In this paper, we only focus on reducing the abstraction gap.

Figure 1 gives a general overview of the two kinds of reduction. In this figure, we assume that a given collection of product variants provide a set of features $\{F1, F2, \ldots, F13\}$ and contain a set of classes $\{C1, C2, \ldots, C47\}$. These features and classes are grouped into minimal disjoin sets at feature and source code levels. VSM is used to compute textual similarity between classes while FCA is used to cluster together similar classes. Each cluster is interpreted as a *code-topic*. LSI is used to link features and their corresponding *code-topics*. After determining the *code-topics* corresponding to each feature, we easily determine classes that implement each feature by decomposing each *code-topic* to its classes. In our approach, a code-topic may be associated to more than one feature.

3 Code-Topics Identification based on Hierarchical Clustering

Clustering, in general, is the division of objects into groups of similar objects. Each group, called a cluster, consists of objects that are similar amongst themselves and dissimilar to objects of other groups [8]. Clustering approaches are classified into hierarchical or non-hierarchical. Hierarchical clustering methods are further categorized into agglomerative (AHC for short) and divisive. Below, we present how AHC is used to identify code-topics.

3.1 Code-Topic Identification as a Partitioning Problem

According to our definition of the *code-topic*, the content of a *code-topic* matches a set of classes. Therefore, in order to determine a set of classes that can belong to a *code-topic*, it is important to formulate the code-topic identification as a partitioning problem. The input is a set of classes (C). This set could be classes of the common partition or classes of any minimal disjoint set (see Figure 1). The output is a set of *code-topics* (T) of C. $C = \{c_1, c_2, ..., c_n\}$ and $T(C) = \{T_1, T_2, \ldots, T_k\}$ where: (1) c_i is a class belonging to C; (2) T_i is a subset of C; (3) k is the number of identified *code-topics*; (4) $T(C)$ does not contain empty elements: $\forall T_i \in T(C), T_i \neq \Phi$; (5) The union of all $T(C)$ elements is equal to C: $\bigcup_{i=1}^{k} T_i = C$.

For a given set of classes, we cluster them based on the strength of the relationship between the classes. In our case, this relationship refers to textual similarity. We use VSM to compute this similarity. We create a document for each class containing a list of terms extracted from all the 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 is treated as a query. The cosine similarity equation (refer to equation 1) represents a fitness function to decide which class documents belong to a cluster.

$$F(d_j, q) = \frac{\sum_{i=0}^{n} w_{i,j} w_{i,q}}{\sqrt{\sum_{i=0}^{n} w_{i,j}^2}\sqrt{\sum_{i=0}^{n} w_{i,q}^2}} \qquad (1)$$

Where `F` is the fitness function with value in the range [0,1]. *d_j* and *q* are class and query vector documents respectively while *w* is a term weight. In our case, we try to maximize the fitness value for each cluster where similar classes use similar vocabulary.

3.2 Building a Hierarchy of Clusters

AHC groups similar classes that use similar vocabulary together and aggregate them into clusters. The basis for clustering classes is the strength of the relationship between them. The previously defined fitness function is used to measure this strength. AHC relies on a series of successive binary mergers, initially of individual class documents and later of clusters formed during the previous stages. We obtain from these binary mergers a single cluster dendrogram (dendgr) that contains a set of nested clusters. Figure 2 shows an example of dendrogram tree. At the lowest level, each class is in its own unique cluster. At the highest level, all classes belong to the same cluster. The internal nodes represent new clusters formed by merging the clusters that appear as their children in the tree.

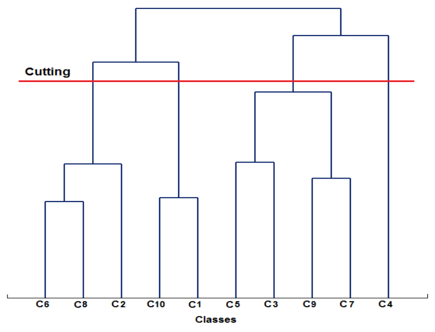


Figure 2. An example of dendrogram tree.

3.3 Selection Candidate Code-Topics

Breaking the dendrogram tree based on predefined criteria allows to group classes into clusters. Each cluster can be a candidate *code-topic*. Therefore, we must select the breaking point to obtain *code-topics*. This selection is performed by an algorithm based on a depth-first search (refer to algorithm 1). This algorithm takes as input the dendrogarm tree and returns a set of clusters. We interpret these clusters as *code-topics*. For each node in the dendrogram (starting from the root), we compare the fitness value of the focused node and its sons (d_j and q in equation 1 become cluster nodes). If the fitness value of the focused node is less than the average of the fitness values of its two sons, then the algorithm continues on to the next son nodes. Otherwise, the focused node is identified as a *code-topic*, added to the *code-topics(T)* and the algorithm computes the next node in the stack. In this way, the most relevant *code-topics* will be identified as the traversal continues.

Algorithm 1: CodeTopicDendrogramTraversal

```
Input: $Dendrogram(dendgr)$
Output: $Code\text{-}Topics(T)$
stack $traversedClusters$
$push(traversedClusters, dendgr)$
while ($|traversedClusters| > 0$) do
    $parent \leftarrow pop(traversedClusters)$
    $son1 \leftarrow getSon1Cluster(parent, dendgr)$
    $son2 \leftarrow getSon2Cluster(parent, dendgr)$
    $avg \leftarrow average(F(son1), F(son2))$
    if ($F(parent) > (avg)$) then
        $add(parent, T)$
    else
        $push(son1, traversedClusters)$
        $push(son2, traversedClusters)$
    end
end
return $T$
```

3.4 Code-Topics for Locating Feature Implementations

After identifying code-topics, we apply LSI (as shown in Figure 1) to link mandatory features and their possible corresponding code-topics, as well as to link each minimal disjoint set of optional features and their possible corresponding code-topics. LSI is applied by following the steps described in section 2.1, but we build LSI's corpus and queries as follows. LSI's corpus consists of code-topic documents, which each one corresponds to a code-topic. Each document consists of terms extracted from identifiers of corresponding code-topic's classes. LSI takes code-topic and feature documents as input. Then, LSI measures the similarity between the code-topics and features using the cosine similarity. This returns a list of code-topics, ordered by their cosine similarity values against each feature. The retrieved code-topics should have a cosine similarity value greater than or equal to *0.70*, where this value represents the most widely used threshold for the cosine similarity [5].

After linking each feature with all its corresponding code-topics, we can easily link each feature with its classes by decomposing each code-topic to its classes. For instance, if the feature *f1* is linked to two code-topics: *topic1*= c1, c4 and *topic2*= c7, c6, by decomposing these topics into their classes; we can find that *f1* is implemented by five classes c1, c4, c7, c6.

4 Experimental Evaluation

4.1 Case Studies and Experiments Setting

To validate our approach, we have applied it to a collection of seven variants of a large-scale system, ArgoUML-SPL[1] modeling tool, and five variants of a small-scale system, MobileMedia[2].

The ArgoUML-SPL is a Java open-source which supports all standard UML 1.4 diagrams. The ArgoUML-SPL's products are generated from the same framework so that products that share the same features also share the same code. The selected products differ in terms of provided features but they support all UML diagram features. These features are implemented by source code classes. To establish ground truth links between features and their classes in order to evaluate our proposal, we compare code classes of two generated products; one of them provides all features while the other provides all features except the focused one. The obtained classes represent the real implementation of the focused feature. We repeat this process for all ArgoUML-SPL's features. MobileMedia is a JAVA open source that manipulates multimedia on mobile devices. In our study, we have considered and analyzed variants that only implement features as classes. To establish ground truth links between features and their source code classes, we analyze manually the source code.

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

[2]Available at http://www.ic.unicamp.br/ tizzei/mobilemedia/

Table 1. Precision, recall and F-measure of AHC against FCA for ArgoUML-SPL and MobileMedia.

ArgoUML-SPL

	Precision		Recall		F-measure	
K	AHC	FCA	AHC	FCA	AHC	FCA
0.1	52%	70%	99%	40%	68%	51%
0.2	52%	57%	99%	9%	68%	16%
0.3	52%	57%	98%	5%	68%	9%
0.4	52%	62%	98%	4%	68%	8%
0.5	52%	57%	96%	2%	67%	3%

MobileMedia

	Precision		Recall		F-measure	
K	AHC	FCA	AHC	FCA	AHC	FCA
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%

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. The proper way to make such a choice is an open issue in the literature. Too many term-topics leads to the association of irrelevant terms and too few term-topics leads to loosely relevant terms. We are unable to use a fixed number of term-topics (*#term-topics*) because we have different sizes of code-topic documents. Therefore, we use a factor K between 0.1 and 0.5. *#term-topics* is equal to $(K \times D_{dim})$, where D_{dim} is the document dimensionality of the term-document matrix that is generated by LSI.

4.2 Results and Discussion

Table 1 summarizes precision, recall and F-measure results of locating all features of ArgoUML-SPL and MobileMdia by using AHC and FCA to identify code-topics.

On a large-scale system (ArgoUML-SPL), we notice that AHC significantly improves the recall values with a minor decrease in the precision compared to FCA. This improvement in recall is due to the fact that AHC identifies *code-topics* by determining a set of clusters so that classes of each cluster are similar amongst themselves and dissimilar to classes of other clusters. Regarding FCA, it identifies *code-topics* by determining a set of clusters in which all cluster's members are similar to each other but it doesn't consider similarity between clusters (refer to [4]). This means that FCA computes the textual similarity only among classes while AHC computes the similarity not only among classes but also among clusters. Therefore, the number of *code-topics* obtained by FCA is higher than AHC (423, 17 respectively). Identifying a small number of *code-topics* means that each code-topic document contain more relevant source code information. This is allow to do better textual matching with feature descriptions, and hence the recall. Regarding the minor decreasing in the precision, this is due to the fact that AHC depends a lot on VSM compared with FCA. AHC uses VSM to compute similarity between classes and clusters while FCA uses VSM to compute similarity only between classes. This means that the number of false-positive links in the case of AHC is higher than FCA because VSM retrieve false-positive links which leads to impression. F-measure results refer to AHC achieve a better compromise between precision and recall than FCA.

On a small system (MobileMedia), it is observed that AHC and FCA produce the same precision, recall and F-measure results for the following reasons. Firstly, most minimal disjoint sets of optional features consist of only one feature, and hence their corresponding minimal disjoint sets of classes contain only the implementation of that feature, no more and no less). Thus, in this case we do not require LSI and *code-topics*. Secondly, MobileMedia's features are implemented by a small number of classes, sometimes by only two classes. These classes have little information that hinders building *code-topics*.

5 Related Work

The approach was proposed by Kuhn et al. [6] is the closet to ours. They proposed an approach to identify *linguistic topics* from object-oriented source code. Their approach relied on LSI to compute similarity among given set of methods, classes or packages. Then, AHC was used to cluster similar elements together as *linguistic topics*. The clusters retrieved by their approach are not necessarily domain concepts (i.e., features), but rather code-oriented topics. In our approach, we identify feature-oriented topics where these topics are identified from a set of classes that implement features.

Maskeri et al. [9] identify business topics from source code by using Latent Dirichlet Allocation (LDA). Their interpretation for a *topic* is a set of semantically related linguistic terms identified from identifiers names and comments. Kawaguchi et al. [10] proposed an automatic categorization method for a large collection of software systems. They use linguistic information in source code for identifying categories (topics) from open source repositories (e.g., *SourceForge*). A *category* is a cluster of related identifiers. Our approach differs from the works of *Maskeri et al.* and *Kawaguchi et al.* in two ways. Firstly, topics in their approaches are clusters of terms while code-topics in our approach are clusters of software artifacts (classes). Secondly, topics retrieved by their approach are code-oriented topics while we identify feature-oriented topics, as they are identified from source code classes that implement features (and only features).

6 Conclusion

In this paper, we have proposed to combine IR and AHC to improve the effectiveness of IR-based feature location in a collection of product variants. This improvement involves reducing the abstraction gap between feature and source code levels by introducing the concept of *code-topic* as an intermediate level. We have compared between two algorithms to identify code-topics: AHC and FCA. In our experimental evaluation using two different case studies, we showed that AHC significantly increases the recall of IR-based feature location with a minor decrease of precision compared to FCA.

References

[1] X. Yinxing, X. Zhenchang, and J. Stan, "Understanding feature evolution in a family of product variants," *Reverse Engineering, Working Conference on*, vol. 0, pp. 109–118, 2010.

[2] 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.

[3] 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.

[4] H. Eyal-Salman, A.-D. Seriai, and C. Dony, "Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval," ser. IRI'13. California, USA: IEEE, 2013, pp. 209–216.

[5] 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.

[6] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, Mar. 2007.

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

[8] P. Berkhin, "A survey of clustering data mining techniques," *Grouping Multidimensional Data*, pp. 25–71, 2006.

[9] G. M. Rama, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent dirichlet allocation." in *ISEC*, 2008, pp. 113–120.

[10] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "Mudablue: An automatic categorization system for open source repositories." in *APSEC*, 2004, pp. 184–193.