Documenting the Mined Feature Implementations from the Object-oriented Source Code of a Collection of Software Product Variants

R. AL-msie'deen[1], A.-D. Seriai[1], M. Huchard[1], C. Urtado[2], and S. Vauttier[2]

[1]LIRMM / CNRS & Montpellier 2 University, France, {al-msiedee, seriai, huchard}@lirmm.fr
[2]LGI2P / Ecole des Mines d'Alès, Nîmes, France, {Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

Abstract

Companies often develop a set of software variants that share some features and differ in other ones to meet specific requirements. To exploit existing software variants and build a Software Product Line (SPL), a Feature Model (FM) of this SPL must be built as a first step. To do so, it is necessary to mine optional and mandatory features in addition to associating the FM with its documentation. In our previous work, we mined a set of feature implementations as identified sets of source code elements. In this paper, we propose a complementary approach, which aims to document the mined feature implementations by giving them names and descriptions, based on the source code elements that form feature implementations and use-case diagrams of software variants. The novelty of our approach is that it exploits commonality and variability across software variants, at feature implementations and use-cases levels, to run Information Retrieval methods in an efficient way. To validate our approach, we applied it on Mobile media and ArgoUML-SPL case studies. The results of this evaluation showed that most of the features have been documented correctly.

Keywords*: Software variants, Software Product Line, Feature documentation, Code comprehension, Formal Concept Analysis, Relational Concept Analysis, Use-case diagram, Latent Semantic Indexing.*

1 Introduction

Software variants often evolve from an initial product, developed for and successfully used by the first customer. These product variants usually share some common features and differ regarding others. As the number of features and the number of software variants grows, it is worth re-engineering them into a Software Product Line (SPL) for systematic reuse [1]. The first step towards re-engineering software variants into SPL is to mine the Feature Model (FM) of these variants. To obtain such a FM, common and optional features for software variants have to be identified. This consists in identifying among source code elements, groups of such elements that implement candidate features and associating them with their documentation (*i.e.*, a feature name and description). In our previous work [2], we proposed an approach for feature mining from the object-oriented source code of software variants (REVPLINE approach[1]). REVPLINE allows us mining of functional features as a set of Source Code Elements (SCEs) (*e.g.*, package, class, attribute, method or method body elements).

To assist a human expert to document the mined feature implementations, we propose an automatic approach which associates names and descriptions using source code elements of feature implementations and use-case diagrams of software variants. Compared with existing work that documents source code (*cf.* section 6), the novelty of our approach is that we exploit commonality and variability across software variants at feature implementation and use-case levels, to apply Information Retrieval (IR) methods in an efficient way. Considering commonality and variability across software variants enables us to cluster the use-cases and feature implementations into *disjoint* and *minimal* clusters based on Relational Concept Analysis (RCA); where each cluster is disjoint from the others and consists of a minimal subset of feature implementations and their corresponding use-cases. Then, we use Latent Semantic Indexing (LSI) to define a similarity measure that enables us to identify which use-cases characterize the name and de-

[1]REVPLINE stands for RE-engineering Software Variants into Software Product Line.

scription of each feature implementation by using Formal Concept Analysis (FCA).

The remainder of this paper is structured as follows: Section 2 briefly presents the background. Section 3 shows an overview of the feature documentation process. Section 4 presents the feature documentation process step by step. Section 5 describes the experimentation. Section 6 discusses the related work. Finally, section 7 concludes and provides perspectives for this work.

2 Background

This section provides a glimpse on FCA, RCA and LSI. It also shortly describes the example that illustrates the remaining sections of the paper.

2.1 Formal and Relational Concept Analysis

FCA [3] is a classification technique that extracts a partially ordered set of concepts (the concept lattice) from a dataset composed of objects described by attributes (the formal context). A concept is composed of two sets: an object set called the concept's extent and an attribute set called the concept's intent. The extent is the maximal set of objects which share the maximal set of attributes of the intent (*cf.* Section 4.3). We use the concepts of the AOC-poset, namely the concepts that introduce at least one object or one attribute. The interested reader can find more information about our use of FCA in [2].

RCA [4] is an iterative version of FCA in which the objects are classified not only according to the attributes they share, but also according to the relations between them (*cf.* Section 4.1). Data are encoded into a *Relational Context Family* (RCF), which is a pair (K, R), where K is a set of formal (object-attribute) contexts $K_i = (O_i, A_i, I_i)$ and R is a set of relational (object-object) contexts $r_{ij} \subseteq O_i \times O_j$, where O_i (domain of r_{ij}) and O_j (range of r_{ij}) are the object sets of the contexts K_i and K_j, respectively (*cf.* Table 1). A RCF is used in an iterative process to generate, at each step, a set of concept lattices. Firstly, concept lattices are built, using the formal contexts only. Then, in the following steps, a scaling mechanism translates the links between objects into conventional FCA attributes and derives a collection of lattices whose concepts are linked by relations (*cf.* Figure 2). The interested reader can find more information about RCA in [4]. For applying FCA and RCA we used the Eclipse eRCA platform[2].

2.2 Latent Semantic Indexing

LSI is an advanced Information Retrieval (IR) method [1]. LSI assumes that software artifacts can be regarded as textual documents. Occurrences of terms are extracted from the documents in order to calculate similarities between them and then to classify together a set of similar documents as related to a common concept (*cf.* Section 4.2). The heart of LSI is the singular value decomposition technique. This technique is used to mitigate noise introduced by stop words (like "the", "an", "above") and to overcome two issues arising in natural languages processing: *synonymy* and *polysemy*. The effectiveness of IR methods is usually measured by metrics including *recall*, *precision* and *F-measure*. In our context, for a given use-case (query), recall is the percentage of correctly retrieved feature implementations (documents) to the total number of relevant feature implementations, while precision is the percentage of correctly retrieved feature implementations to the total number of retrieved feature implementations. F-Measure defines a trade-off between precision and recall, so that it gives a high value only in cases where both recall and precision are high. All measures have values in [0%, 100%]. If recall equals 100%, all relevant feature implementations (documents) are retrieved. However, some retrieved feature implementations might not be relevant. If precision equals 100%, all retrieved feature implementations are relevant. Nevertheless, relevant feature implementations might not be retrieved. If F-Measure equals 100%, all relevant feature implementations are retrieved. However, some retrieved feature implementations might not be relevant. The interested reader can find more information about our use of LSI in [2].

2.3 The Mobile Tourist Guide Example

We consider in this example four software variants of a Mobile Tourist Guide (MTG) application. These applications enable users to inquire about some tourist information on mobile devices. MTG_1 supports core MTG functionalities: *view map*, *place marker on a map*, *view direction*, *launch Google map* and *show street view*. MTG_2 has the core MTG functionalities and a new functionality called *download map from Google*. MTG_3 has the core MTG functionalities and a new functionality called *show satellite view*. MTG_4 supports *search for nearest attraction*, *show next attraction* and *retrieve data* functionalities, together with the core ones.

3 The Feature Documentation Process

Our goal is to document the mined feature implementations by using the use-case diagrams of these variants. In our work, we rely on the same assumption as in the work of [5] stating that each use-case represents a feature. The feature documentation process aims at identifying which

[2]The eRCA : `http://code.google.com/p/erca/`

use-cases characterize the name and description of each feature implementation. We rely on lexical similarity to identify the use-cases that characterize the name and description of feature implementations. The performance and efficiency of the IR technique depends on the size of the search space. In order to apply LSI, we take advantage of the commonality and variability between software variants to group feature implementations and the corresponding use-cases in the software family into disjoint, minimal clusters (*e.g.*, Concept_1 of Figure 2). We call each disjoint minimal cluster a *Hybrid Block* (HB). After reducing the search space to a set of hybrid blocks, we rely on textual similarity to identify, from each hybrid block, which use-cases depict the name and description of each feature implementation.

For a product variant, our approach takes as inputs the set of use-cases that documents the variant and the set of mined feature implementations that are produced by REVPLINE. Each use-case is identified by its name and description. This information represents domain knowledge that is usually available from software variants documentation (*i.e.*, requirement model). In our work, the use-case description consists of a short paragraph in a natural language. Our approach provides as its output a name and description for each feature implementation based on a use-case name and description. Each use-case is mapped into a functional feature thanks to our assumption. If two or more use-cases have a relation with the same feature implementation, we consider them all as the documentation for this feature implementation.

Figure 1 shows an overview of our feature documentation process. The first step of this process aims at identifying hybrid blocks based on RCA (*cf.* Section 4.1). In the second step, LSI is applied to determine the similarity between use-cases and feature implementations (*cf.* Section 4.2). This similarity measure is used to identify use-case clusters based on FCA. Each cluster identifies the name and description for feature implementation (*cf.* Section 4.3).

4 Feature Documentation Step by Step

In this section, we describe the feature documentation process step by step. According to our approach, we identify the feature name and its description in three steps as detailed in the following.

4.1 Identifying Hybrid Blocks of Use-cases and Feature Implementations via RCA

We use the existing use-case diagrams of software variants to document the feature implementations mined from those variants. In order to apply LSI in an efficient way, we need to reduce the search space for use-cases and feature implementations. Starting from existing feature implementations and use-cases, these elements are clustered into disjoint minimal clusters (*i.e.*, hybrid blocks) to apply LSI. The search space is reduced based on the commonality and variability of software variants. RCA is used to cluster: the use-cases and feature implementations common to all software variants; the use-cases and feature implementations that are shared by a set of software variants, but not all variants; the use-cases and feature implementations that are held by a single variant.

A RCF for feature documentation is automatically generated from use-case diagrams and the mined feature implementations associated with software variants[3]. The RCF corresponding to our approach contains two formal contexts and one relational context, as illustrated in Table 1. The first formal context represents the *use-case diagrams*. The second formal context represents *feature implementations*. In the formal context of *use-case diagrams*, objects are use-cases and attributes are software variants. In the formal context of *feature implementations*, objects are feature implementations and attributes are software variants. The relational context (*i.e.*, *appears-with*) indicates which use-case appears in the same software variants as feature implementations.

For the RCF presented in Table 1, a close-up view of two lattices of the Concept Lattice Family (CLF) is represented in Figure 2. As an example of hybrid block we can see in Figure 2 a set of use-cases (in the extent of $Concept\_1$ of the *Use_case_Diagrams* lattice) that always appear with a set of feature implementations (in the extent of $Concept\_6$ of the *Feature_Implementations* lattice). As shown in Figure 2, RCA allows us to reduce the search space by exploiting

[3]Source code : `https://code.google.com/p/rcafca/`

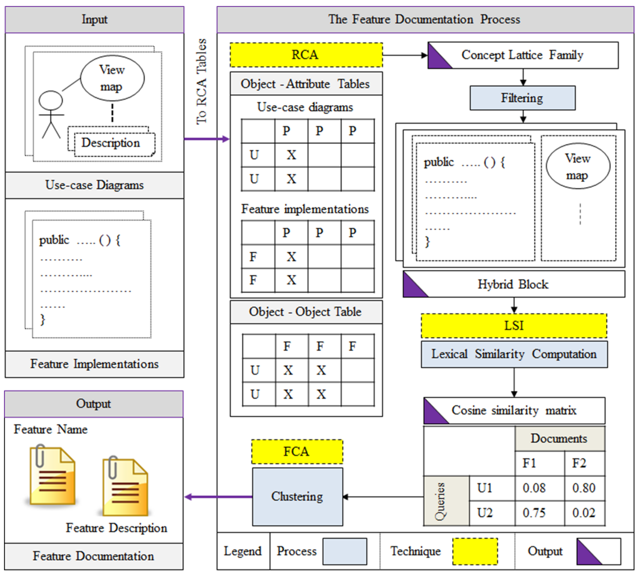

Figure 1: The feature documentation process.

Table 1: The RCF for features documentation.

Use_case_Diagrams	MTG_1	MTG_2	MTG_3	MTG_4	Feature_Implementations	MTG_1	MTG_2	MTG_3	MTG_4
View Map	✗	✗	✗	✗	Feature Implementation_1	✗	✗	✗	✗
Launch Google Map	✗	✗	✗	✗	Feature Implementation_2	✗	✗	✗	✗
View Direction	✗	✗	✗	✗	Feature Implementation_3	✗	✗	✗	✗
Show Street View	✗	✗	✗	✗	Feature Implementation_4	✗	✗	✗	✗
Place Marker on Map	✗	✗	✗	✗	Feature Implementation_5	✗	✗	✗	✗
Download Map		✗			Feature Implementation_6		✗		
Show Satellite View			✗		Feature Implementation_7			✗	
Show Next Attraction				✗	Feature Implementation_8				✗
Search For nearest attraction				✗	Feature Implementation_9				✗
Retrieve Data				✗	Feature Implementation_10				✗

Relational context: appears-with	Feature Implementation_1	Feature Implementation_2	Feature Implementation_3	Feature Implementation_4	Feature Implementation_5	Feature Implementation_6	Feature Implementation_7	Feature Implementation_8	Feature Implementation_9	Feature Implementation_10
View Map	✗	✗	✗	✗	✗					
Launch Google Map	✗	✗	✗	✗	✗					
View Direction	✗	✗	✗	✗	✗					
Show Street View	✗	✗	✗	✗	✗					
Place Marker on Map	✗	✗	✗	✗	✗					
Download Map						✗				
Show Satellite View							✗			
Show Next Attraction								✗	✗	✗
Search For Nearest Attraction								✗	✗	✗
Retrieve Data								✗	✗	✗

commonality and variability across software variants. In our work, we are filtering CLF to get a set of hybrid blocks from bottom to top[4]. Figure 2 shows an example of hybrid block (the dashed block).

4.2 Measuring the Lexical Similarity Between Use-cases and Feature Implementations via LSI

Based on the previous step, each hybrid block consists of a set of use-cases and a set of feature implementations. We need to identify from each hybrid block which use-cases characterize the name and description of each feature implementation. To do so, we use textual similarity between use-cases and feature implementations. This similarity measure is calculated using LSI. We rely on the fact that a use-case corresponding to the feature implementation is supposed to be lexically closer to this feature implementation than to the other feature implementations. Similarity between use-cases and feature implementations in the hybrid blocks is computed in three steps as detailed below.

[4]Source code : `https://code.google.com/p/fecola/`

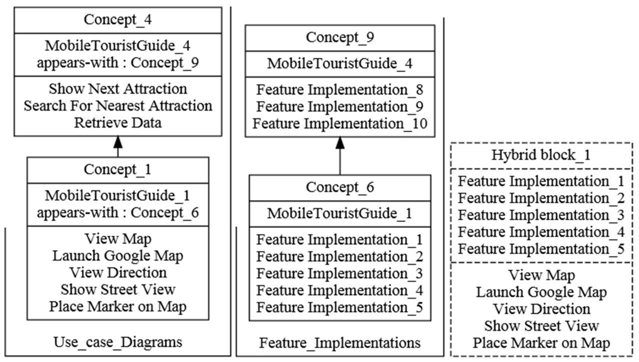

Figure 2: Parts of the CLF deduced from Table 1.

4.2.1 Building the LSI Corpus

In order to apply LSI, we build a corpus that represents a collection of documents and queries. In our work, each *use-case* name and description in the hybrid block represents a *query* and each *feature implementation* represents a *document*. This document contains all the segments of SCE names as a result of splitting words into terms using the Camel-case technique. Regardless of word location (first, middle or last) in the SCE name, we store all words in the document. For example, for the SCE name *ManualTestWrapper* all words are important: *manual*, *test* and *wrapper*. We apply the same technique to all feature implementations. Our approach creates a query for each use-case. This query contains the use-case name and its description. We apply the same process to all use-cases. To be processed, documents and queries must be normalized as follows: stop words, articles, punctuation marks, or numbers are removed; text is tokenized and lower-cased; text is split into terms; stemming is performed (*e.g.*, removing word endings); terms are sorted alphabetically. We use WordNet[5] to do some simple preprocessing (*e.g.*, stemming and removal of stop words). The most important parameter of LSI is the number of term-topics (*i.e.*, *k-Topics*) chosen. A term-topic is a collection of terms that co-occur often in the documents of the corpus, for example {*user, account, password, authentication*}. In our work, the number of *k-Topics* is equal to the number of feature implementations for each corpus.

4.2.2 Building the Term-document and the Term-query Matrices for each Hybrid Block

All hybrid blocks are considered and the same processes are applied to them. The *term-document matrix* is of size $m \times n$, where m is the number of terms extracted from feature implementations and n is the number of feature implementations (*i.e.*, documents) in a corpus. The matrix values indicate the number of occurrences of a term in a document, according to a specific weighting scheme. In our work, terms are weighted with the *TF-IDF* function (the most common weighting scheme) [1] . The *term-query matrix* is of size $m \times n$, where m is the number of terms that are extracted from use-cases and n is the number of use-cases (*i.e.*, queries) in a corpus. An entry of term-query matrix refers to the weight of the i^{th} term in the j^{th} query.

[5]`http://wordnet.princeton.edu/`

4.2.3 Building the Cosine Similarity Matrix

Similarity between use-cases and feature implementations in each hybrid block is described by a *cosine similarity matrix* which columns (documents) represent vectors of feature implementations and rows (queries) vectors of use-cases. The textual similarity between documents and queries is measured by the cosine of the angle between their corresponding vectors [2].

4.3 Identifying Feature Name via FCA

Based on the cosine similarity matrix we use FCA to identify, from each hybrid block of use-cases and feature implementations, which elements are similar. To transform the (numerical) cosine similarity matrices into (binary) formal contexts, we use a 0.7 threshold (after having tested many threshold values). This means that only pairs of use-cases and feature implementations having a calculated similarity greater than or equal to 0.70 are considered similar.

For example, for the hybrid block *Concept_1* of Figure 2 the number of term-topics of LSI is equal to 5. In the formal context associated with this hybrid block, the use-case *"Launch Google Map"* is linked to the feature implementation *"Feature Implementation_1"* because their similarity equals 0.86, which is greater than the threshold. However, the use-case *"View Direction"* and the feature implementation *"Feature Implementation_5"* are not linked because their similarity equals 0.10, which is less than the threshold. The resulting AOC-poset is composed of concepts which *extent* represents the use-case name and *intent* represents the feature implementation.

For the MTG example, the AOC-poset of Figure 3 shows five non comparable concepts (that correspond to five distinct features) mined from a single hybrid block (*Concept_1* from Figure 2). The same feature documentation process is used for each hybrid block.

5 Experimentation

To validate our approach, we ran experiments on two Java open-source softwares: Mobile media software variants (small systems) [6] and ArgoUML-SPL (large systems) [7]. We used 4 variants for Mobile media, 10 for ArgoUML.

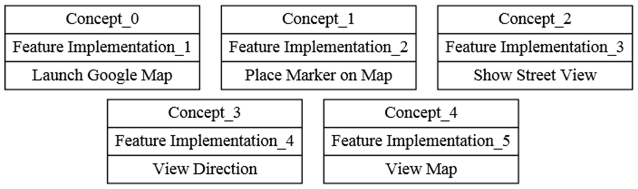

Figure 3: The documented features from *Concept_1*.

The advantage of having two case studies is that they implement variability at different levels: class and method levels. In addition, these case studies are well documented and their feature implementations, use-case diagrams and FMs are available for comparison of our results and validation of our proposal[6]. Table 2 summarizes the obtained results.

Table 2: Features documented from case studies.

#	Feature Name	Hybrid block #	k-Topics	Evaluation Metrics		
				Recall	Precision	F-Measure
Mobile Media						
1	Delete Album	HB_1	4	100%	100%	100%
2	Delete Photo	HB_1	4	100%	50%	66%
3	Add Album	HB_1	4	100%	100%	100%
4	Add Photo	HB_1	4	100%	50%	66%
5	Exception handling	HB_2	1	100%	100%	100%
6	Count Photo	HB_3	3	100%	50%	66%
7	View Sorted Photos	HB_3	3	100%	50%	66%
8	Edit Label	HB_3	3	100%	100%	100%
9	Set Favourites	HB_4	2	100%	50%	66%
10	View Favourites	HB_3	2	100%	50%	66%
ArgoUML-SPL						
1	Class diagram	HB_1	1	100%	100%	100%
2	Logging	HB_2	2	100%	50%	66%
3	Cognitive support	HB_2	2	100%	100%	100%
4	Deployment diagram	HB_3	1	100%	100%	100%
5	Collaboration diagram	HB_4	2	100%	50%	66%
6	Sequence diagram	HB_4	2	100%	50%	66%
7	State diagram	HB_5	1	100%	100%	100%
8	Activity diagram	HB_6	2	100%	100%	100%
9	Use case diagram	HB_6	2	100%	100%	100%

For the two case studies presented, we observe that the *recall* values are 100% of all the features that are documented. The recall values are an indicator for the efficiency of our approach. The values of precision are between [50% - 100%], which is high. F-Measure values rely on precision and recall values. The values of F-Measure are high too, between [66% - 100%] for the documented features. Results show that recall value in all cases is 100% and value of precision either 100% or 50% it is because of the similarity threshold (0.70) in addition, this result is due to search space reduction. In most cases, the contents of hybrid blocks are in the range of $[1-4]$ use-cases and feature implementations. Another reason for this good result is that a common vocabulary is used in the use-case descriptions and feature implementations, thus lexical similarity was a suitable tool. In our work we cannot use a fixed number of topics for LSI because we have hybrid blocks (clusters) with different sizes.

The column (*k-Topics*) in Table 2 represents *the number of term-topics*. All feature names produced by our approach, in the column (*Feature Name*) of Table 2, represent the names of the use cases. For example, in the FM of Mobile media [6] there is a feature called *sorting*. The name proposed by our approach for this feature is *view sorted photos* and its description is "*the device sorts the photos based on the number of times photo has been viewed*".

[6]Case studies and code : `http://www.lirmm.fr/CaseStudy`

As a *limitation* to our approach, developers might not use the same vocabularies to name SCEs and use-cases across software variants. This means that lexical similarity may not be reliable (or should be improved with other techniques) in all cases to identify the relationship between use-cases and feature implementations. Furthermore, using FCA as clustering technique has also limits. FCA deals with binary formal context. This affects the quality of the result, since a similarity value of 0.99 (*resp.* 0.69) is treated as a similarity value of 0.70 (*resp.* 0). Selecting the appropriate number of dimensions (K) for the LSI representation is an open research question.

6 Related Work

Most existing approaches are designed to extract labels, names, topics or identify code to use-case traceability links in a single software system. In the context of feature documentation, most existing approaches manually assign feature names (without any description) to feature implementations. Conversely our approach is designed to automatically assign a name and a description to each feature implementation in a set of software variants based on several techniques (FCA, RCA and LSI). Feature documentation is inferred from use case names and descriptions.

Ziadi *et al.* [8] propose an approach to identify features across software variants. In their work, they manually create feature names. In our previous work [2], we manually propose feature names for the mined feature implementations. Braganca and Machado [5] describe an approach for automating the process of transforming UML use-cases into FMs. In their work, each use-case is mapped to a feature. The identification of relationships (*i.e.*, traceability links) between use-case diagrams and source code for a single software is the subject of the work by Grechanik *et al.* [9]. Kuhn *et al.* [10] present a lexical approach that uses the log-likelihood ratios of word frequencies to automatically provide labels for components of single software. Xue *et al.* [1] propose an automatic approach to identify the code-to-feature traceability link for a collection of software product variants.

7 Conclusion and Perspectives

In this paper, we proposed an approach for documenting the mined feature implementations of a set of software variants. We exploit commonalities and variabilities between software variants at feature implementation and use-case levels to apply IR methods in an efficient way in order to automatically document the mined features. We have implemented our approach and evaluated its results on two case studies. The results of this evaluation showed that most of the features were documented correctly. Regarding future work, we plan to use the mined and documented features to build automatically the relations between the features of a FM (*i.e.*, reverse engineering FMs).

Acknowledgments: This work has been supported by project CUTTER ANR-10-BLAN-0219.

References

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

[2] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing," in *SEKE*, 2013, pp. 244–249.

[3] B. Ganter and R. Wille, *Formal concept analysis - mathematical foundations*. Springer, 1999.

[4] M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev, "Relational concept discovery in structured datasets," *Ann. Math. Artif. Intell.*, vol. 49, no. 1-4, pp. 39–76, 2007.

[5] A. Bragança and R. J. Machado, "Automating mappings between use case diagrams and feature models for software product lines," in *SPLC*. IEEE, 2007, pp. 3–12.

[6] L. P. Tizzei, M. O. Dias, C. M. F. Rubira, A. Garcia, and J. Lee, "Components meet aspects: Assessing design stability of a software product line," *Information & Software Technology*, vol. 53, no. 2, pp. 121–136, 2011.

[7] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *CSMR*, 2011, pp. 191–200.

[8] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane, "Feature identification from the source code of product variants," in *CSMR*, 2012, pp. 417–422.

[9] M. Grechanik, K. S. McKinley, and D. E. Perry, "Recovering and using use-case-diagram-to-source-code traceability links," in *ESEC/SIGSOFT FSE*, 2007, pp. 95–104.

[10] A. Kuhn, "Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code," in *MSR*, M. W. Godfrey and J. Whitehead, Eds. IEEE, 2009, pp. 175–178.