Documenting Features Mined 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, Montpellier, France*
{*Al-msiedee, Abdelhak.Seriai, Huchard*}*@lirmm.fr*
[2] *LGI2P / Ecole des Mines d'Alès, Nîmes, France*
{*Christelle.Urtado, Sylvain.Vauttier*}*@mines-ales.fr*

***Abstract*—Companies may develop a set of similar software variants, even though they do not necessarily have a Software Product Line (SPL). They start constructing, marketing and selling individual products, which they then modify, customize and extend for different customers. To exploit existing software variants for reuse (build a SPL) and facilitate its maintenance, a feature model of these software variants must be built as a first step. To do so, it is necessary to mine optional and mandatory features in addition to associating it with its documentation. In our previous work, we mined a set of feature implementations as source code units from software variants. In this paper, we propose the approach of documenting the mined features by giving names and descriptions, based on the feature implementations and use-case diagrams of software variants. The novelty of our approach is that we exploit commonality and variability across software variants, at feature implementation and use-cases levels, to apply Information Retrieval (IR) methods in an efficient way. To validate our approach, we applied it on Mobile media, Health Complaint-SPL and ArgoUML-SPL case studies. The results of this evaluation showed that most of the features had been documented correctly.**

***Keywords*-Feature, Documentation, Comprehension, Software Product Variants, Use-case Diagram, Formal Concept Analysis, Relational Concept Analysis, Latent Semantic Indexing, Software Product Line.**

I. Introduction

Software variants often evolve from an initial product, developed for and successfully used by the first customer. Mobile Media [1] is an example of such product evolution. 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 reengineering them into a Software Product Line (SPL) for systematic reuse [2]. A SPL is a family of related software variants that share some common artefacts (*e.g.*, source code, use-case diagram, design documents, *etc.*) and differ in relation to others [3]. SPLs are usually described with a *de facto* standard formalism called Feature Model (FM) [4]. Feature model defines all the valid feature configurations. The first step towards the migration of software variants into SPL is to mine the FM of these variants. For obtaining such a FM, common and optional features for software variants have to be identified. This corresponds with identifying the implementation of each feature and associating its documentation (*i.e.*, a feature name and description). In our previous works [5] [4], we proposed an approach for feature mining from the object-oriented source code of software variants (REVPLINE approach[1]). REVPLINE allows the mining of functional features as a set of Object-oriented Units (*i.e.*, OUs) (*e.g.*, package, class, attribute, method or method body elements). The implementation of each feature may correspond to a huge number of OUs; the mined feature must be documented. Therefore, we propose in this paper a technique to document the mined feature implementations. The goal of this documentation is to reflect feature roles at the domain level. Additionally, for purposes of constructing an FM and reusing existing features in other software, each feature implementation that is presented to the human user must have a meaningful name. In addition, feature documentation is needed in order to understand existing software variants and facilitate their maintenance. We rely on the feature implementations and use-case diagrams of software variants to propose naming and descriptions of the identified feature implementation. Compared with existing works proposing to document source code [6] [7] [8] [5] *etc.* (*cf.* Section VII), the novelty of our approach is that we exploit commonality and variability across software product variants at feature implementation and use-case levels, to apply IR methods in an efficient way in order to document the mined implementation for each feature. Considering commonality and variability across software product variants allows us to cluster the use-cases and feature implementations into a *disjoint minimal partition* based on Relational Concept Analysis (RCA); where each partition is *disjoint* and consists of a *minimal* subset of feature implementations and the corresponding use-cases through exploiting the commonalities and variabilities across software variants. Then, we use Latent Semantic Indexing (LSI) to define a similarity measure that enables us to identify which use-

[1]REVPLINE stands for REengineering Software Variants into Software Product Line

cases characterize the name and description of each feature implementation.

The remainder of this paper is structured as follows: Section II briefly presents the background needed to understand our proposal. Then section III shows an overview of the feature documentation process. Next, section IV presents the feature documenting process step by step. Section V describes the experiments that were conducted to validate our proposal. Then section VI discusses threats to the validity of our approach. Section VII discusses the related work; while finally, section VIII concludes and provides perspectives for this work.

II. Background

This section provides a glimpse of Formal Concept Analysis (FCA), Relational Concept Analysis (RCA), Latent Semantic Indexing (LSI) and use-case diagram. It also shortly describes the example that illustrates the remaining sections of the paper.

A. Formal Concept Analysis (FCA) and Relational Concept Analysis (RCA)

Galois lattices and concept lattices [9] are core structures of a data analysis framework (Formal Concept Analysis) for extracting an ordered set of concepts from a dataset, called a formal context, composed of objects described by attributes. The purpose of FCA is to automatically find groups of objects (or entities) that share a common group of attributes. Table I shows an example of a formal context for describing animals by their properties. In the formal

Table I: A formal context for describing animals.

	flying	nocturnal	feathered	migratory	duck-billed	web
flying squirrel	×					×
bat	×	×				
ostrich			×			
flamingo	×		×	×		
sea-gull	×		×		×	

context (*cf.* Table I) the objects are animals and attributes are animal's properties. Figure 1 shows the Hasse diagram of the concept lattice structuring our animals. In this diagram, extents and intents are presented in a simplified form: removing up-down inherited attributes and down-up inherited objects. In our approach, we will consider the AOC-poset (for Attribute-Object-Concept poset), which is the sub-order of $(\mathcal{C}_K, \leq_s)$, restricted to object-concepts and attribute-concepts. AOC-posets scale much better than lattices. For our example, it would correspond to the concept lattice of Figure 1 deprived of `Concept_0`, `Concept_3` and `Concept_4`. The AOC-poset (*cf.* Figure 1) is composed of concepts whose *extent* represents animal names and *intent* represents animal properties. The interested reader can find more information about FCA in [9].

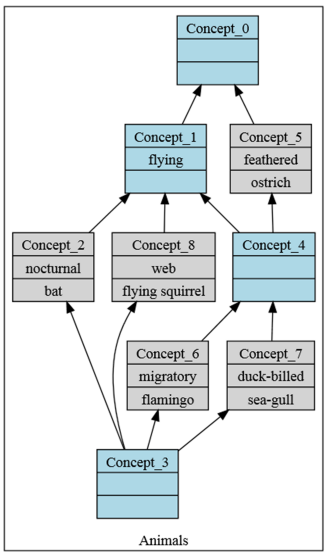

Figure 1: The concept lattice for the formal context of Table I.

RCA [10] 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. 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. The Relational Context Family (RCF) corresponding to our example contains four formal contexts (*motion, animals, places, food*) and four relational context (*lives, moves, offers and eatBy*), illustrated in Table II.

Table II: The RCF for animals.

Motion
fly
swim

Animals
eagle
bat
catfish

Places
mountain
cave
river

Food
mouse
insect
fish

lives	mountain	cave	river
eagle	×		
bat		×	
catfish			×

moves	fly	swim
eagle	×	
bat	×	
catfish		×

offers	mouse	insect	fish
mountain	×		
cave		×	
river			×

eatBy	eagle	bat	catfish
mouse	×		
insect		×	
fish			×

An RCF is used in an iterative process to generate, at each step, a set of concept lattices [10]. 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). For example, the existential scaled relation (which we will use in this paper) captures the following information: if an object o_s is linked to another object o_t,

then in the scaled relation, o_s will receive relational attributes associated to concepts, which group o_t with other objects. This is used to form new groups, for example the group (*cf.* $Concept\_5$ in Figure 2) of *Food*, which are eaten by a group of animals (such are grouped in $Concept\_1$ of the *Animals*). The steps are repeated until the lattices become stable (*i.e.*, when no more new concepts are generated). For RCF in Table II, four lattices of the concept lattice family are represented in Figure 2. The interested reader can find more information about RCA in [10]. For applying FCA and RCA we used the Eclipse eRCA platform[2].

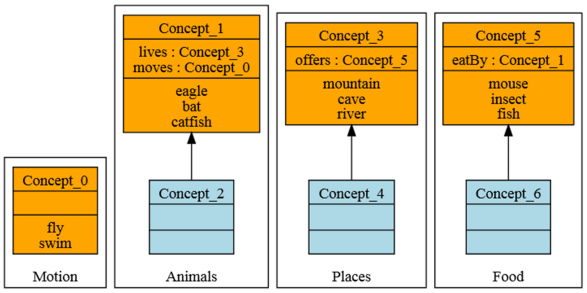

Figure 2: The concept lattice family of RCF in Table II.

B. Latent Semantic Indexing (LSI)

Information Retrieval (IR) refers to techniques that compute textual similarity among different documents. The textual similarity is computed based on the occurrences of terms within documents. If two documents share a large number of terms, those documents are considered to be similar. Different IR techniques have been proposed, such as Latent Semantic Indexing (LSI) and Vector Space Model (VSM), to compute textual similarity [11]. LSI is an advanced IR method. The core of LSI is Singular Value Decomposition (SVD) technique. This technique is used to mitigate noise introduced by stop words (*e.g.*, "the", "an", "above") and to overcome two classical problems arising in natural language processing: synonymy and polysemy [3]. The LSI procedure can be divided into three steps:

- A corpus of documents is built after pre-processing such as stop word removal and stemming performing.
- The term-document matrix is created by using the corpus. The matrix's columns correspond to the corpus documents and rows represent terms that are extracted from the documents. The matrix values indicate the number of occurrences of a term in a document according to a specific weighting scheme.
- Similarity among documents is calculated using cosine similarity matrix.

The retrieval performance of the LSI model depends on the term weighting, which indicates the nature of the relationship between a term and a document. The cosine of the angle between the document and the query vector is used as the numeric similarity between the vectors[3]. The effectiveness of IR methods is usually measured by their *recall*, *precision* and *F-measure*. For a given query, recall is the percentage of correctly retrieved links to the total number of relevant links, while precision is the percentage of correctly retrieved links to the total number of retrieved links. 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, 1]. If recall equals 1, all relevant links are retrieved. However, some retrieved links might not be relevant. If precision equals 1, all retrieved links are relevant. Nevertheless, relevant links might not be retrieved. If F-Measure equals 1, all relevant links are retrieved. However, some retrieved links might not be relevant [3].

C. Use-case diagram through our illustrative example

In our work, we rely on the same assumption used in the work of [12] stating that each use-case represents a feature. The use-case diagrams are used to document the mined feature implementation by giving name and description according to the use-case name and its description. Use-case diagrams appear early within a UML-based development, structured over the concepts of actors and use-cases to capture the user requirements of an application [13]. According to [12], a use-case is the "specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system". Figure 3 gives an example of two use-case diagrams for two applications of Mobile Tourist Guide (MTG). As an illustrative example, we consider four software variants of MTG. These applications allow a user to inquire about some tourist information through the mobile device. 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.

III. The Feature Documentation Process

Our goal is to document the mined feature implementations from a collection of software variants. Based on existing use-case diagrams of software variants, we document the mined features by combining both use-cases and feature implementations. Feature documentation processes targets to identify which use-cases characterize the name

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

[3]For the purpose of our approach, we developed our LSI tool. Available at https://code.google.com/p/lirmmlsi/

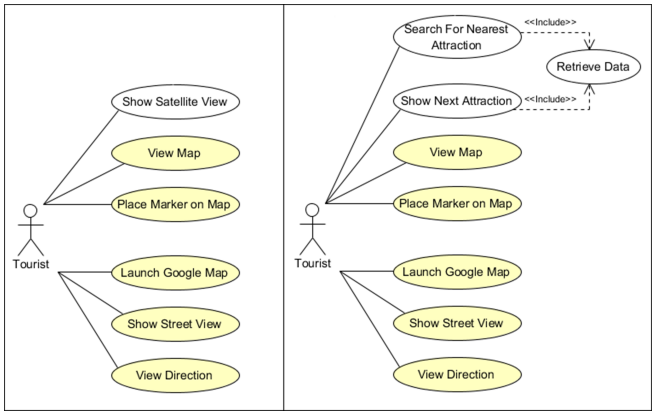

Figure 3: The use-case diagram for MTG_3 and MTG_4.

and description of each feature implementation. We rely on lexical similarity to identify those use-cases that characterize the name and description of each feature implementation. IR technique's performance and efficiency depends on the size of the search space. In order to apply LSI, we take advantage of the commonalities and variabilities across software variants for group feature implementations and the corresponding use-cases in the software family into disjoint, minimal partitions. As an example of disjoint minimal partition; the use-cases and feature implementations that are common to all software variants will cluster together as one cluster (*i.e.*, minimal search space). We called each disjoint minimal partition a *hybrid block*. After reducing search space into a set of hybrid blocks, we rely on the textual similarity to identify, from each hybrid block, which use-cases depict the name and description of each feature implementation. After identifying the textual similarity between use-cases and feature implementations, we group those elements as a set of clusters based on the textual similarity.

For a product variant, our approach takes as input a set of use-cases that the product variant supports and a set of mined feature implementations. Each use-case is identified by its name and description. A use-case description is given in a natural language. This information about the use-case represents a 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. For example, *retrieve data* use-case of Figure 3 is described in the following paragraph, "*the tourist can retrieve information and a small image of the attraction through his/her mobile phone. In addition, the tourist can store the current view of the map to the mobile phone*". Our approach gives as output for each feature implementation, a name and description based on the use-case name and description. Each use-case is mapped into a functional feature based on our assumption. In the case of there being two or more use-cases have a relation with the single feature implementation, we consider all relevant use-cases as documentation for this feature implementation.

The feature documentation process takes the variants' use-cases and the mined feature implementations as its inputs. The first step of this process aims at identifying hybrid blocks based on RCA (*cf.* Section IV-A). In the second step, we rely on LSI to determine the similarity between use-cases and feature implementations (*cf.* Section IV-B). This similarity measure is used to identify sets of clusters based on FCA. Each cluster identifies the name and description for feature implementation (*cf.* Section IV-C). Figure 4 shows our feature documentation process.

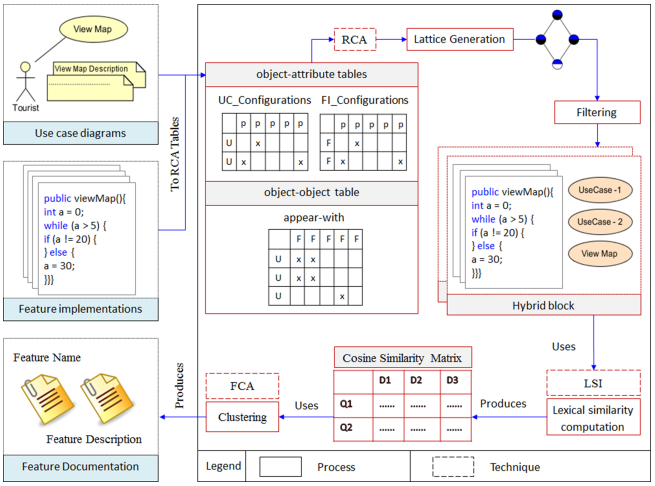

Figure 4: The feature documentation process.

In our work we consider RCA and FCA as a clustering method. RCA links each cluster of use-cases with the corresponding cluster of feature implementations by exploiting commonalities and variabilities across software variants. The concept lattice family facilitates the comprehension of the clusters and their relations and provides a better visualization. LSI had positive results in addressing comprehension and maintenance tasks, such as feature location [5], recovery of traceability links between source code and different software artifacts [14] [2] [15], naming of software components [6], and labelling of software source code [16].

IV. 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: i) identifying hybrid blocks of use-cases and feature implementations via RCA, ii) measuring the lexical similarity between use-cases and feature implementations via LSI and iii) identifying the feature name and its description via FCA as detailed in the following.

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

We utilise the existing use-case diagrams of software variants to document the mined feature implementations from those variants. In order to apply the 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 we need to cluster these elements into disjoint minimal partitions (*i.e.*, hybrid blocks) to apply the LSI. Reducing the search space is based on the commonalities and variabilities of software variants. To do so, there is a need to exploit commonalities and variabilities across software variants. To achieve this objective, the RCA completes the first step of the feature documentation process. Using RCA to reduce the search space, the common use-cases and feature implementations among all software variants appear together as one cluster. The use-cases and feature implementations that are shared among a set of software variants, but not all variants, appear together as one cluster. Also the use-cases and feature implementations that are unique for a single variant appear together as one cluster.

The Relational Context Family (RCF) corresponding to our approach contains two formal contexts and one relational context, illustrated in Table III. The first formal context represents the *use-case diagrams*. The second formal context represents *feature implementations*. In the formal context of *use-case diagrams*, the objects are use-cases and attributes are software variants. In the formal context of *feature implementations*, the objects are feature implementations and attributes are software variants. The relational context (*i.e.*, *appear-with*) indicates which use-case appears with which feature implementation across software variants.

Table III: The RCF for documenting features.

Use_case_Diagrams	MobileTouristGuide_1	MobileTouristGuide_2	MobileTouristGuide_3	MobileTouristGuide_4
View Map	x	x	x	x
Launch Google Map	x	x	x	x
View Direction	x	x	x	x
Show Street View	x	x	x	x
Place Marker on Map	x	x	x	x
Download Map		x		
Show Satellite View			x	
Show Next Attraction				x
Search For Nearest Attraction				x
Retrieve Data				x

Feature_Implementations	MobileTouristGuide_1	MobileTouristGuide_2	MobileTouristGuide_3	MobileTouristGuide_4
Feature Implementation_1	x	x	x	x
Feature Implementation_2	x	x	x	x
Feature Implementation_3	x	x	x	x
Feature Implementation_4	x	x	x	x
Feature Implementation_5	x	x	x	x
Feature Implementation_6		x		
Feature Implementation_7			x	
Feature Implementation_8				x
Feature Implementation_9				x
Feature Implementation_10				x

Relational context: appear-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	x	x	x	x	x					
Launch Google Map	x	x	x	x	x					
View Direction	x	x	x	x	x					
Show Street View	x	x	x	x	x					
Place Marker on Map	x	x	x	x	x					
Download Map						x				
Show Satellite View							x			
Show Next Attraction								x	x	x
Search For Nearest Attraction								x	x	x
Retrieve Data								x	x	x

For RCF in Table III, two lattices of the concept lattice family are represented in Figure 5. As an example of the hybrid block in Figure 5, we can see a set of use-cases (*cf.* $Concept\_1$ of the *Use_case_Diagrams* lattice) always appears with a set of feature implementations (*cf.* $Concept\_6$ of the *Feature_Implementations* lattice) based on software configurations at use-case and feature implementation levels. As shown in Figure 5, RCA allows us to reduce the search space by exploiting commonalities and variabilities across software variants.

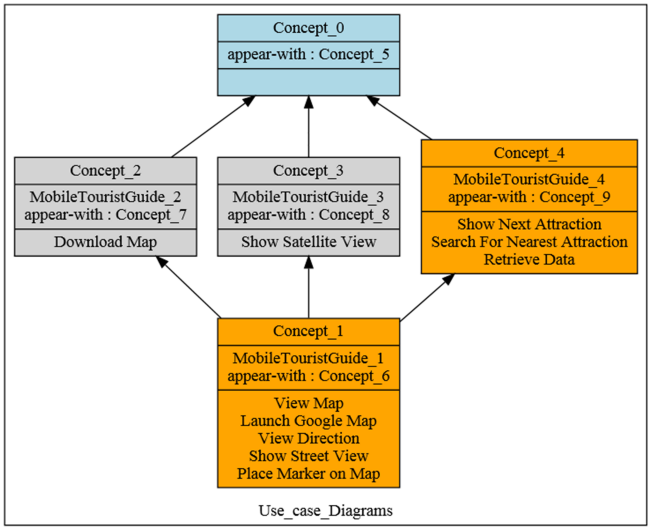

Figure 5: The concept lattice family of RCF in Table III.

In our work, we are exploring concept lattice family and filtering them to get a set of hybrid blocks from bottom to top[4] (*i.e.*, from the largest search space to smaller search space). RCF for feature documentation is generated automatically from use-case diagrams and the mined feature

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

implementations of software variants[5].

B. 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 for each feature implementation. To do so, we utilise the 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 lexically closer to this feature implementation than to the rest of feature implementations.

To compute similarity between use-cases and feature implementation in the hybrid blocks, we proceed in three steps: building the LSI corpus, building the term-document matrix and the term-query matrix for each hybrid block and, at lastly, building the cosine similarity matrix.

1) Building the LSI corpus: In order to apply LSI, we build a corpus that represents a collection of documents and queries. In our case, each use-case name and description in the hybrid block represents a query and each feature implementation represents a document (*cf.* Figure 6).

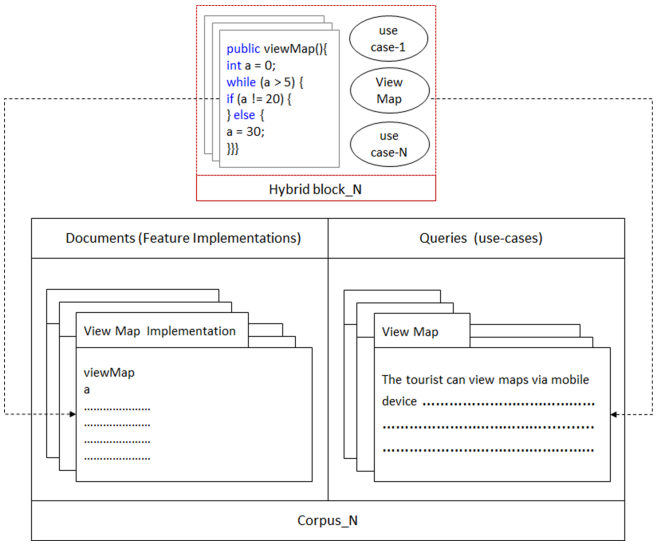

Figure 6: Constructing a raw corpus from Hybrid block.

Each feature implementation contains a set of *Object-oriented Units* (OUs) such as packages, classes, attributes, methods or method body elements. Each feature implementation is represented by one document. Each document contains all OUs names that form this feature implementation. In our work, we store the complete name of the OU, due to the importance of a complete OU name. Regardless of word location (first, middle or last) through OU name, we store all words in the document. For example, for the OU name *ManualTestWrapper* all words are important {*manual, test and wrapper*}. The same for all OU names (*e.g.*, package, class, attribute, method, local variable, method invocation or attribute access), the complete name is stored in the document. Each feature implementation is mapped into a document with all OU names. For the query, our approach creates a document for each use-case. This document contains the use-case name and its description. To be processed, the document and query must be normalized (*e.g.*, all capitals turned into lower case letters, articles, punctuation marks or numbers removed). The normalized document is generated by analyzing the OU names of feature implementation. All OU names are split into terms and at last, word stemming is performed. The same procedure is followed to manipulate the use-case and its description to get the query document. 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*}. Due to the nature of language use, the terms that form a topic are often semantically related. In our work, the number of k-Topics are equal to the number of feature implementations for each corpus.

2) Building the term-document and the term-query matrices for each hybrid block: All hybrid blocks are considered and the same processes 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 (*cf.* Table IV). 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 i^{th} term in the j^{th} query. Terms for both matrices are the same because they are extracted from the same hybrid block (*cf.* Table IV). Table IV shows the term-document and the term-query matrices of the hybrid block of *Concept_1* from Figure 5.

Table IV: The term-document and the term-query matrices of *Concept_1* from Figure 5.

	Feature Implement_1	Feature Implement_2	Feature Implement_3	Feature Implement_4	Feature Implement_5
device	1.0	0.0	0.0	0.0	1.0
direction	0.0	0.0	0.0	6.0	0.0
google	1.0	0.0	0.0	0.0	0.0
launch	4.0	0.0	0.0	0.0	0.0
map	1.0	2.0	0.0	0.0	4.0
marker	0.0	6.0	0.0	0.0	0.0
mobile	1.0	0.0	0.0	0.0	1.0
place	0.0	3.0	0.0	0.0	0.0
show	0.0	0.0	2.0	0.0	0.0
street	0.0	0.0	5.0	0.0	0.0
tourist	1.0	1.0	1.0	1.0	1.0
view	0.0	0.0	1.0	2.0	5.0

The term-document matrix

	Launch Google Map	Place Marker on Map	Show Street View	View Direction	View Map
device	1.0	0.0	0.0	0.0	1.0
direction	0.0	0.0	0.0	8.0	0.0
google	3.0	0.0	0.0	0.0	0.0
launch	3.0	0.0	0.0	0.0	0.0
map	2.0	2.0	1.0	1.0	5.0
marker	0.0	3.0	0.0	0.0	0.0
mobile	1.0	0.0	0.0	0.0	1.0
place	0.0	3.0	0.0	0.0	0.0
show	0.0	0.0	3.0	0.0	0.0
street	0.0	0.0	5.0	0.0	0.0
tourist	1.0	1.0	1.0	1.0	1.0
view	0.0	0.0	1.0	3.0	5.0

The term-query matrix

[5]https://code.google.com/p/rcafca/

In the term-document matrix, the *direction* term appears 6 times in the document *Feature Implementation_4*. In the term-query matrix, the *direction* term appears 8 times in the query *view direction*. The term-document matrix shows all terms of corpus and gives the number of occurrences of each term in each document.

3) Building the cosine similarity matrix: Similarity between use-cases and feature implementations in each hybrid block is described by a cosine similarity matrix whose columns represent vectors of feature implementations and rows represent vectors of use-cases: documents as columns and queries as rows. The textual similarity between documents and queries is measured by the cosine of the angle between their corresponding vectors [11]. The *k-Topics* for LSI in our approach are equal to the number of feature implementations in each hybrid block. In this example, *K* value is equal to 5. Table V shows the cosine similarity matrix of *Concept_1* (*i.e.*, hybrid block) from Figure 5.

Table V: The cosine similarity matrix of *Concept_1*.

	Feature Implementation_1	Feature Implementation_2	Feature Implementation_3	Feature Implementation_4	Feature Implementation_5
Launch Google Map	0.861933577	0.0137010	0.0	0.0	0.152407
Place Marker on Map	0.01114798	0.9480070	0.0	0.0	0.085939
Show Street View	0.004088722	0.0051128	0.98581691	0.00571	0.070920
View Direction	0.00296571	0.0037085	0.0069484	0.999139665	0.108597
View Map	0.114676597	0.0627020	0.039159941	0.070025418	0.994111

C. Identifying Feature Name and Description via FCA

Based on the cosine similarity matrix we use FCA to identify, from each hybrid block of use-cases and feature implementations, which are similar elements. To transform the (numerical) cosine similarity matrices of the previous step into (binary) formal contexts, we use 0.70 as a threshold. 0.70 is currently used for cosine similarity (after testing many thresholds). 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. Table VI shows the formal context obtained by transforming the cosine similarity matrix corresponding to the hybrid block of *Concept_1* from Figure 5.

Table VI: Formal context of *Concept_1*.

	Feature Implementation_1	Feature Implementation_2	Feature Implementation_3	Feature Implementation_4	Feature Implementation_5
Launch Google Map	x				
Place Marker on Map		x			
Show Street View			x		
View Direction				x	
View Map					x

In this formal context, 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 (*cf.* Figure 7) is composed of concepts whose *extent* represents the use-case name and *intent* represents the feature implementation.

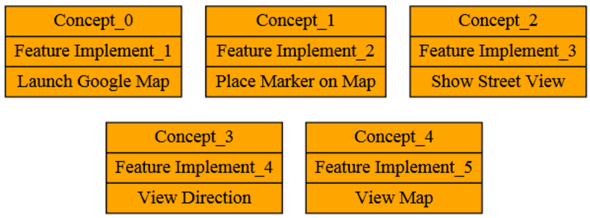

Figure 7: The documented features from *Concept_1*.

For the MTG example, the AOC-poset of Figure 7 shows five concepts (that correspond to five distinct features) mined from a single hybrid block (*Concept_1* from Figure 5). The same feature documentation process is used for each hybrid block.

V. Experimentation

This section presents the case studies in which we apply our approach. In addition, it presents the feature documentation results.

A. Case studies

To validate our approach, we ran experiments on three Java open-source softwares: Mobile Media software variants[6], Health complaint-SPL[7] and ArgoUML-SPL[8]. We used 8 variants for Mobile Media, 5 variants for Health complaint-SPL and 6 variants for ArgoUML-SPL. The selected software variants contain all the features of each family. Mobile Media variants [1], Health complaint-SPL[9] and ArgoUML-SPL [17] are well documented. Their source code, use-case diagram and feature model are available for comparison with our results and validation of our proposal[10]. Feature implementation sizes vary from one case study to another (Mobile media (small), Health complaint-SPL (medium) and ArgoUML-SPL (large)). For example, the size of the *cognitive support* feature of ArgoUML case study is 16,319 lines of code (LOC), while the size of *play music* feature of Mobile media case study is 709 LOC. Every case study

[6]http://www.ic.unicamp.br/~tizzei/mobilemedia/
[7]http://ptolemy.cs.iastate.edu/design-study/
[8]http://argouml-spl.tigris.org/
[9]http://www.ic.unicamp.br/~tizzei/phc/jss2013/
[10]Feature implementations, use-case diagrams and use-case descriptions for each case study are available on our web site http://www.lirmm.fr/CaseStudy

is different from other case studies in terms of the number of use-cases and the mined feature implementations. The Mobile media case study is classified as *software product variants* while the Health complaint-SPL and ArgoUML-SPL are classified as a *software product line*.

Mobile media is a Java-based open source application that manipulates photo, music, and video on mobile devices, such as mobile phones [18]. Table VII shows and describes Mobile media variants by their features.

Table VII: Mobile media software variants.

Mobile Media	Delete album	Delete photo	Create album	Create photo	View photo	Exception handling	Edit photo label	Sorting	Count Photo	Set favourites	View favourites	Copy photo	Send photo	Receive photo	Play music	Play video	Capture video
S1	x	x	x	x	x												
S2	x	x	x	x	x	x											
S3	x	x	x	x	x	x	x	x	x								
S4	x	x	x	x	x	x	x	x	x	x	x						
S5	x	x	x	x	x	x	x	x	x	x	x	x					
S6	x	x	x	x	x	x	x	x	x	x	x	x	x	x			
S7	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
S8	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Health complaint is a Java-based open source web application that manages health related records and complaints. Health complaint allows citizens to report complaints related to public health via the Internet. The types of complaints encompass food, animal, drug, and special complaints. Table VIII shows and describes Health complaint-SPL software applications by their features.

Table VIII: Health complaint-SPL software applications.

Health Complaint-SPL	Specify complaint	Food complaint	Animal complaint	Special complaint	Drug complaint	Update employee	Update complaint	Update health unit	Login	Change logged	Register table	Register new employee	Query information	Find information	Receive alert
S1	x	x	x	x	x										
S2	x	x	x	x	x	x	x	x	x	x					
S3	x	x	x	x	x	x	x	x	x	x	x	x			
S4	x	x	x	x	x								x	x	x
S5	x	x	x	x	x						x	x	x	x	x

ArgoUML is a Java-based, open source tool, widely used for designing systems in UML. The FM for the ArgoUML-SPL as manually designed by the authors of ArgoUML-SPL is presented in [17]. It contains 10 features. Table IX shows and describes ArgoUML-SPL software applications by their features.

Table IX: ArgoUML-SPL software applications.

ArgoUML-SPL	Diagrams	Class	Cognitive support	Logging	State	use-case	Deployment	Collaboration	Sequence	Activity
S1	x	x								
S2	x	x	x	x	x	x	x	x	x	x
S3	x	x		x						x
S4	x	x	x		x					
S5	x	x					x		x	
S6	x	x				x		x		

B. Result

Table X summarizes the results obtained from documenting the mined features of Mobile media, Health complaint-SPL and ArgoUML-SPL case studies.

Table X: Features documented from Mobile media, Health complaint-SPL and ArgoUML-SPL case studies.

#	Feature names for each case study	Hybrid block #	k-Topics	U-FI (SV) ††	Evaluation Metrics		
					Recall	Precision	F-Measure
I	Mobile media						
1	Add Album	HB_1	5	95%	100%	50%	66%
2	Add Photo	HB_1	5	79%	100%	50%	66%
3	Delete Album	HB_1	5	93%	100%	50%	66%
4	Delete Photo	HB_1	5	99%	100%	50%	66%
5	View Photo	HB_1	5	99%	100%	100%	100%
6	Capture Media	HB_2	2	81%	100%	100%	100%
7	Play Video	HB_2	2	90%	100%	100%	100%
8	Play Music	HB_3	1	97%	100%	100%	100%
9	Receive Photo	HB_4	2	100%	100%	50%	66%
10	Send Photo	HB_4	2	97%	100%	50%	66%
11	Copy Photo	HB_5	1	94%	100%	100%	100%
12	Count Photo	HB_6	3	92%	100%	100%	100%
13	Edit Label	HB_6	3	98%	100%	100%	100%
14	View Sorted Photos	HB_6	3	91%	100%	100%	100%
15	Exception Handling	HB_7	1	100%	100%	100%	100%
16	Set Favourites	HB_8	2	100%	100%	50%	66%
17	View Favourites	HB_8	2	100%	100%	50%	66%
II	Health Complaint						
1	Animal complaint specification	HB_1	5	92%	100%	100%	100%
2	Drug complaint specification	HB_1	5	99%	100%	100%	100%
3	Food complaint specification	HB_1	5	99%	100%	100%	100%
4	Special complaint specification	HB_1	5	89%	100%	100%	100%
5	Specify complaint	HB_1	5	97%	100%	100%	100%
6	Register tables	HB_2	2	93%	100%	100%	100%
7	Register new employee	HB_2	2	99%	100%	100%	100%
8	Change logged employee	HB_3	5	78%	100%	100%	100%
9	Login	HB_3	5	85%	100%	100%	100%
10	Update complaint	HB_3	5	99%	100%	100%	100%
11	Update employee	HB_3	5	97%	100%	100%	100%
12	Update health unit	HB_3	5	97%	100%	100%	100%
13	Find Information	HB_4	3	92%	100%	100%	100%
14	Query information	HB_4	3	95%	100%	100%	100%
15	Receive alerts via feeds	HB_4	3	82%	100%	100%	100%
III	ArgoUML						
1	Class diagram	HB_1	2	95%	100%	100%	100%
2	Diagrams	HB_1	2	89%	100%	100%	100%
3	Login	HB_2	2	99%	100%	100%	100%
4	Activity diagram	HB_2	2	99%	100%	100%	100%
5	Cognitive support	HB_3	2	85%	100%	50%	66%
6	State diagram	HB_3	2	85%	100%	50%	66%
7	use-case diagram	HB_4	2	99%	100%	100%	100%
8	Deployment diagram	HB_4	2	99%	100%	100%	100%
9	Collaboration diagram	HB_5	2	90%	100%	50%	66%
10	Sequence diagram	HB_5	2	93%	100%	50%	66%

U-FI (SV) ††: The similarity value between use-case and the relevant feature implementation.

Throughout our work, each use-case name and description represents a query and each feature implementation represents a document. This can also be vice versa. In practice, both cases give the same result in terms of the retrieved elements. For the three case studies presented, we can see that the *recall* values are 100% of all features that are documented. The recall values are an indicator for the accuracy of our approach. From recall values, we can see that for each use-case, there is a correctly retrieved feature implementation. *Precision* values also are high. The values of precision are between [50% - 100%]. *F-Measure* values also are high. F-Measure values rely on precision and recall values. The values of F-Measure are between [66% - 100%] for the documented features. In this experimentation, we use different case studies. Fortunately, in the three case studies, there is a common vocabulary between use-cases and feature implementations. In our approach, each use-case is mapped into a functional feature. Fortunately, there is no

limitation of our approach in cases with more than one use-case mapped into a feature. Based on the textual similarity, we consider the related use-cases as a description of this feature implementation.

Our results can be interpreted on the basis that reducing the search space for use-cases and feature implementations across software variants is the reason behind this result. In most cases, the contents of hybrid blocks are in the range of [1 - 5] use-cases and feature implementations. A further reason for this result is the common vocabulary used in the use-case descriptions and feature implementations, which helped us to have good results from the lexical similarity, for those features in which precision value is not 100%, such as *sequence* and *collaboration* diagrams. The reason behind this result is that a feature implementation can have a common vocabulary with many use-case descriptions.

The column (*k-Topics*) in Table X represents the *number of topics*. We use it to determine the number of topics in each hybrid block. In our approach, the number of topics is equal to the number of feature implementations in each hybrid block. The column (*Feature names for each case study*) in Table X represents the proposed feature names for each case study. The proposed feature name for each feature implementation has the same use-case name.

Comparing our results with the feature names that are included in the FM, we found that our results are very close to the feature names in FM. For example, in the FM of Mobile media [18] there is a feature called *sorting*, the proposed name of this feature, by our approach, is *view sorted photos* and its description is "*the device sorts the photos based on the number of times photo has been viewed*".

In our work, we represent the similarity values between the use-cases and feature implementations. The results are represented as a directed graph. Use-case and feature implementation are represented as vertices and the similarity links as edges. The degree of similarity appears along the edges of the graph[11].

VI. Threats to validity

As a limitation of our approach, developers might not use the same vocabularies to name OUs and use-cases across software variants. This means that lexical similarity may be not reliable (or should be improved with other techniques) in all cases to identify the relationship between use-case and feature implementation. Furthermore, there is a limitation using FCA as clustering technique. FCA deals with binary formal context (1, 0). When we transform the (numerical) cosine similarity matrices into (binary) formal contexts, we use a threshold. So if the similarity value between query and document is greater than or equal the 0.70 the two documents are considered similar. By contrast, if the similarity value is less than the threshold (*i.e.*, 0.69) the two documents are considered not similar. FCA deals with discrete values (0, 1). This affects on the quality of the result, where the similarity value 0.99 is equal to 0.70 and 0.69 is equal to 0.

[11] www.lirmm.fr/~seriai/encadrements/theses/rafat/index.php?n=T.V

VII. Related Work

Ziadi *et al.* [8] propose an approach to identify features across software variants. In their work they propose manually creating the feature names. In our previous works [5] [4] [3], we presented an approach for feature mining in a collection of software product variants. We manually associated feature names to the feature implementations, based on the study of the content of each implementation and on our knowledge on software.

An inclusive survey about approaches recovering feature-to-code traceability links in single software are proposed in [15]. The identification of relationship (*i.e.*, traceability links) between use-case diagrams and source code of single software is the subject of the work by Grechanik *et al.* [14]. In our work, we identify the relationship between feature implementations and the use-case diagrams of a collection of software variants. Xue *et al.* [2] propose an automatic approach to identify the traceability link between a given collection of features and a given collection of source code variants. They thus consider feature descriptions as an input.

Kuhn *et al.* [6] present a lexical approach that uses the log-likelihood ratios of word frequencies to automatically provide labels for components of single software. Their approach can be applied i) to compare components with each other, ii) to compare a component against a normative corpus, and iii) to compare different revisions of the same component. Kebir *et al.* [7] propose an approach to identify components from object-oriented source code of single software. Their approach proposed allocating names to the components based on the class names. Their work identifies component names in three steps: extracting and tokenizing class names from the identified cluster, weighting words and constructing the component name by using the strongest weighted tokens. De Lucia *et al.* [16] propose an approach for source code labelling, based on IR technique, to identify relevant words in the source code of single software. They applied various IR methods to extract terms from class names by means of some representative words, with the aim of facilitating their comprehension or simply to improve visualization. Falleri *et al.* [19] proposed a wordNet-like approach to extract the structure of single software by using the relationships among identifier names (*e.g.*, packages, classes, methods, variables, *etc.*). The approach considers Natural Language Processing techniques, which consist of tokenization process (straightforward decomposition technique by word markers, *e.g.*, case changes, underscore, *etc.*), part of speech tagging, and rearranging order of terms by the dominance order of term rules, based on part of speech.

Davril *et al.* [20] present an approach to constructing FMs from product descriptions. They developed a cluster-naming process that involved selecting the most frequently occurring term from amongst all of the feature descriptors in the cluster using the Stanford Part-of-Speech tagger. Braganca and Machado [12] describe an approach for automating the process of transforming UML use-cases to FMs. Their approach explores the $include$ and $extend$ relationships between use-cases to discover relationships between features. In their work, each use-case is mapped to a feature.

VIII. 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 across software variants at feature implementation and use-case levels to apply IR methods in an efficient way in order to document their features. We have implemented our approach and evaluated its produced results on Mobile media, Health Complaint-SPL and ArgoUML-SPL case studies. The results of this evaluation showed that most of the features were documented correctly. Regarding future work, we plan to use search based algorithms for clustering the use-cases and relevant feature implementation together instead of FCA. Also In addition, we plan to use part of speech tagging to document the mined features directly from feature implementations. Finally, we plan to use the mined and documented features (*i.e.*, mandatory and optional features) to automate the building of the feature model.

References

[1] J. M. Conejero, E. Figueiredo, A. Garcia, J. Hernández, and E. Jurado, "On the relationship of concern metrics and requirements maintainability," *Inf. Softw. Technol.*, vol. 54, no. 2, pp. 212–238, Feb. 2012.

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

[3] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, and S. Vauttier, "Mining features from the object-oriented source code of software variants by combining lexical and structural similarity," in *IRI '13 Conference*, 2013, pp. 586–593.

[4] 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 '13 Conference*, 2013, pp. 244–249.

[5] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. Eyal-Salman, "Feature location in a collection of software product variants using formal concept analysis," in *ICSR '13 Conference*, 2013, pp. 302–307.

[6] A. Kuhn, "Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code," in *MSR '09 Conference*. IEEE Computer Society, 2009, pp. 175–178.

[7] S. Kebir, A.-D. Seriai, S. Chardigny, and A. Chaoui, "Quality-centric approach for software component identification from object-oriented code," in *WICSA-ECSA '12 Conference*. IEEE Computer Society, 2012, pp. 181–190.

[8] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane, "Feature identification from the source code of product variants," in *CSMR '12 Conference*. IEEE Computer Society, 2012, pp. 417–422.

[9] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, 1st ed. Springer-Verlag New York, Inc., 1997.

[10] M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev, "Relational concept discovery in structured datasets," *Annals of Mathematics and Artificial Intelligence*, vol. 49, no. 1-4, pp. 39–76, Apr. 2007.

[11] S. W. Thomas, "Mining software repositories using topic models," in *ICSE '11 Conference*. ACM, 2011, pp. 1138–1139.

[12] A. Braganca and R. J. Machado, "Automating mappings between use case diagrams and feature models for software product lines," in *SPLC '07 Conference*. IEEE Computer Society, 2007, pp. 3–12.

[13] X. Dolques, M. Huchard, C. Nebut, and P. Reitz, "Fixing generalization defects in uml use case diagrams," *Fundam. Inf.*, vol. 115, no. 4, pp. 327–356, Dec. 2012.

[14] M. Grechanik, K. S. McKinley, and D. E. Perry, "Recovering and using use-case-diagram-to-source-code traceability links," in *PESEC-FSE '07*. ACM, 2007, pp. 95–104.

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

[16] A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: Is it worthwhile?" in *ICPC*, 2012, pp. 193–202.

[17] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *CSMR '11 Conference*. IEEE Computer Society, 2011, pp. 191–200.

[18] L. P. Tizzei, M. Dias, C. M. F. Rubira, A. Garcia, and J. Lee, "Components meet aspects: Assessing design stability of a software product line," *Inf. Softw. Technol.*, vol. 53, no. 2, pp. 121–136, Feb. 2011.

[19] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao, "Automatic extraction of a wordnet-like identifier network from software," in *ICPC '10 Conference*. IEEE Computer Society, 2010, pp. 4–13.

[20] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans, "Feature model extraction from large collections of informal product descriptions," in *ESEC/FSE 413*. ACM, 2013, pp. 290–300.