International Journal of Software Engineering and Knowledge Engineering

Automatic documentation of [mined] feature implementations from source code elements and use case diagrams with the REVPLINE approach

R. AL-msie'deen, M. Huchard and A.-D. Seriai
LIRMM / CNRS and Montpellier University
Montpellier - France
{*al-msiedee, huchard, seriai*}*@lirmm.fr*

C. Urtado and S. Vauttier
LGI2P / Ecole des Mines d'Alès
Nîmes, France
{*Christelle.Urtado, Sylvain.Vauttier*}*@mines-ales.fr*

Received (31 August 2014)
Accepted (16 October 2014)

Companies often develop a set of software variants that share some features and differ in other ones to meet specific requirements. To exploit the existing software variants as a Software Product Line (SPL), a Feature Model of this SPL must be built as a first step. To do so, it is necessary to define and document the optional and mandatory features that compose the variants. 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 the use-case diagrams that specify software variants. The novelty of our approach is its use of commonality and variability across software variants, at feature implementation and use-case levels, to run Information Retrieval methods in an efficient way. Experiments on several real case studies (Mobile media and ArgoUML-SPL) validate our approach and show promising results.

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

1. Introduction

Similarly to car developers who propose a full range of cars with common characteristics and numerous variants, software developers may cater to various needs and propose as a result a software family instead of a single product. Such a software family is called a Software Product Line (SPL) [1].

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 grow, it is worth re-engineering them into a SPL for systematic reuse [2].

The first step towards re-engineering software variants into SPL is to mine a Feature Model (FM). To obtain such a FM, the common and optional features that compose these variants have to be identified and documented. This consists in (step 1) identifying, among source code elements, groups that implement candidate features and (step 2) associating them with their documentation (*i.e.*, a feature name and description). In our previous work [3], we proposed an approach for step 1 which consists in mining features from the object-oriented source code of software variants (the REVPLINE approach[a]). REVPLINE mines functional features as sets of Source Code Elements (SCEs) (*e.g.*, packages, classes, attributes, methods or method bodies).

In this article, we address step 2. To assist a human expert in documenting the mined feature implementations, we propose an automatic approach which associates names and descriptions using the source code elements of feature implementations and the use-case diagrams of software variants. Compared with existing work that documents source code (*cf.* Section 2), the novelty of our approach is that we exploit commonality and variability across software variants at feature implementation and use-case levels in order to apply Information Retrieval (IR) methods in an efficient way.

Considering commonality and variability across software variants enables to group use-cases and feature implementations into *disjoint* and *minimal* clusters based on Relational Concept Analysis (RCA). Each cluster consists of a subset of feature implementations and their corresponding use-cases. Then, we use Latent Semantic Indexing (LSI) to measure similarities and identify which use-cases best characterize the name and description of each feature implementation by using Formal Concept Analysis (FCA). In the cases where use case diagrams or documentation are not available, we propose an alternative approach, based on the names of the source code elements that implement features.

The remainder of this paper is structured as follows: Section 2 presents the state of the art and motivates our work. Section 3 briefly describes the technical background which is used in our work. Section 4 outlines the feature documentation process. Section 5 details the feature documentation process based on the use case diagrams of variants step by step. Section 6 presents the feature documentation process based on SCE names. Section 7 reports the experimentation and discusses threats to the validity of our approach. Finally, Section 8 concludes and provides perspectives for this work.

[a] REVPLINE stands for RE-engineering software Variants into a software Product LINE.

2. State of the art

In our approach, we aim at documenting groups of source code elements that are the result of a mining process. These groups of source code elements correspond to feature implementations. The comprehension or documentation of feature implementation is a complex problem-solving task [4]. We consider here that documentating a feature is the process of analyzing the implementation of a feature to provide it with either a name or a more detailed description based on software artifacts such as use-case diagrams or identifier names. Related approaches are documentating the source code of a single software, finding traceability links between source code and documentation, and documentating mined features in software product lines.

This section presents research papers in these three fields. We then conclude this state of the art by a synthesis which introduces the main objectives of our approach.

2.1. *Source code documentation in a single software*

Kebir *et al.* [5] propose to identify components from the object-oriented source code of a single software. Their approach assigns names to the components in three steps: extracting and tokenizing class names from the identified implementation of a component, weighting words and building the component name by using the strongest weighted tokens.

Kuhn [6] presents a lexical approach that uses the log-likelihood ratio of word frequencies to automatically retrieve labels from source code. This approach can be applied to compare several components (*i.e.*, describing their differences as well as their commonalities), a component against a normative corpus (*i.e.*, providing labels for components) and different versions of the same component (*i.e.*, documenting the history of a component). In Kuhn *et al.* [7], information retrieval techniques are used to exploit linguistic information found in source code, such as identifier names and comments, in order to enrich software analysis with the developers' knowledge that is hidden in the code. They introduce semantic clustering, a technique based on LSI and clustering to group source artifacts (*i.e.*, classes) that use similar vocabulary. They call these groups semantic clusters and they interpret them as linguistic topics that reveal the intention of the code. They compare the topics, identify links between them, provide automatically retrieved labels (using LSI again to automatically label the clusters with their most relevant terms). They finally use distribution maps to illustrate how the semantic clusters are distributed over the system. Their work is language independent as it works at the level of identifier names.

De Lucia *et al.* [8] propose an approach for source code labeling, based on IR techniques, that identifies relevant words in the source code of a single software. They apply various IR methods (such as VSM, LSI and Latent Dirichlet Allocation (LDA)) to extract terms from class names by means of some representative words, with the aim of facilitating code comprehension or improving visualization. This work investigates to what extent IR-based source code labeling would identify relevant words in the source code, compared to the words a human would manually select during a program comprehension task.

A technique for automatically summarizing source code by leveraging the lexical and structural information in the code is proposed in Haiduc *et al.* [9]. Summaries are obtained from the content of a document by selecting the most important information in that document. The goal of this approach is the automatic generation of summaries for source code entities.

Falleri *et al.* [10] propose a wordNet-like approach to extract the structure of a single software using relationships among identifier names. The approach considers natural language processing techniques which consist of a tokenization process (straightforward decomposition technique by word markers, *e.g.*, case changes, underscore, *etc.*), part of speech tagging and rearranging order of terms by term dominance order rules based on part of speech information.

Sridhara *et al.* [11] present a novel technique to automatically generate comments for Java methods. They use the signature and body of a method (*i.e.*, method calls) to generate a descriptive natural language summary of the method. The developer remains responsible for verifying the accuracy of generated summaries. The objective of this approach is to ease program comprehension. Authors use natural language processing techniques to automatically generate leading method comments. Studies have shown that good comments help programmers understand quickly what a method does, thus assisting program comprehension and software maintenance.

2.2. ***Source code-to-documentation traceability links***

Grechanik *et al.* [12] propose a novel approach for partially automating the process of recovering traceability links (TLs) between types and variables in Java programs and elements of use-case diagrams (UCDs). Authors evaluate their prototype implementation on open-source and commercial software, and their results suggest that their approach can recover many traceability links with a high automation degree and precision. As UCDs are widely used to describe the functional requirements of software products, these traces help programmers understand the code that they maintain and modify.

Marcus *et al.* [13] use LSI to recover traceability links between source code and documentation. The documentation consists of requirement documents which describe elements of the problem domain such as manuals, design models or test suites. This documentation is supposed to have been written before implementation and does not include any parts of the source code.

Diaz *et al.* [14] capture relationships between source code artifacts to improve the recovery of traceability links between documentation and source code. They extract the author of each source code component and, for each author, identify the "context" she/he worked on. Thus, to link documentation and source code artifacts (*i.e.*, use cases and classes), they compute the similarity between these use cases and the authors' contexts. When retrieving related classes using a standard IR-based approach (*e.g.*, LSI or VSM) they reward all the classes developed by authors whose contexts are most similar to use cases.

Xue *et al.* [2] automatically identify traceability links between a given collection of features and a given collection of source code variants. They consider feature descriptions as an input.

2.3. *Documentation of mined features in SPL*

Braganca and Machado [15] describe an approach for automating the transformation of UML use-cases into FMs. In their work, each use-case is mapped to a feature. Their approach explores the *include* and *extend* relationships between use-cases to discover relationships between features. Their work assumes that the feature name is given by that of the use-case.

Yang *et al.* [16] analyze open source applications for multiple domains with similar functionalities. They propose an approach to recover domain feature models using data access semantics, FCA, concept pruning/merging, structure reconstruction and variability analysis. After concept pruning/merging, analysts examine each of the generated candidate feature (*i.e.*, concept cluster) to evaluate its relevance. Meaningless candidate features are removed, whilst meaningful candidate features are chosen as domain features. Then analysts name each domain feature with the help of the corresponding concept intent and extent. After these manual examination and naming, all domain features bear significant names denoting their business functions.

Paškevičius *et al.* [17] present a framework for an automated derivation of FMs from existing software artifacts (*e.g.*, classes, components, libraries, *etc.*), which includes a formal description of FMs, a program-feature relation meta-model and a method for FM generation based on feature dependency extraction and clustering. FMs are generated as Feature Description Language (FDL) descriptors and as Prolog rules. They focus on reverse engineering of source code to FMs and assume that feature names are provided by that of the class or component.

Ziadi *et al.* [18] propose a semi-automatic approach to identify features from object-oriented source code. Their approach takes the source code of a set of product variants as its input. They *manually* assign names to the identified feature implementations by relying on the feature names that are used in the original FM.

In our previous work [19, 3, 20], we manually propose feature names for the mined feature implementations, based on the code elements of each feature implementation.

Davril *et al.* [21] build FMs from product descriptions. Extracting FMs from these informal data sources includes mining feature descriptions from sets of product descriptions, naming the features in a way that is understandable to human users and then discovering relations between features in order to organize them hierarchically into a comprehensive model. The identified features correspond to clusters of descriptions. Authors propose a method to name a cluster using the most frequently and less verbose occurring phrase in the descriptors of this cluster.

2.4. *Synthesis*

Most existing approaches are designed to extract labels, names, topics or to identify traceability links between code and documentation artifacts in a single software system. To document mined features, most existing approaches manually assign feature names to feature implementations (without any further description) and they often rely on atomic source code element names (*e.g.*, class or component names). The most advanced approach for automatic feature description extraction is that of [21] which works on informal product descriptions.

Our work addresses the problem of documenting features mined from several variants of a software system. Our mined feature implementations are sets of source code elements that are more formal artifacts than the product descriptions used in [21] and give complementary information as compared to use case diagrams [15]. Our approach relies on commonality and variability across the variants to apply information retrieval methods more efficiently than in a single software system. Our input data are the source code and use case diagrams of the variants. We do not consider any prior knowledge about features contrarily to [2]. We aim at automatically assigning a name and a description to each mined feature implementation using several techniques (Formal Concept Analysis, Relational Concept Analysis and Latent Semantic Indexing), whereas several approaches manually assign names [16, 18, 3]. Feature documentation consists in use case names, tokens from the source code elements and use case descriptions.

3. Technical 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.

3.1. *Formal and Relational Concept Analysis*

Formal Concept Analysis (FCA) is a classification technique that takes as an input data sets describing objects and their attributes and extracts concepts that are maximal groups of objects (concept extents) sharing maximal groups of attributes (concept intents) [22]. It has many applications in software engineering [23, 24, 25]. The extracted concepts are linked by a partial order relation, which represents concept specialization, as an order which has a lattice structure (called the concept lattice). In the concept lattice, each attribute (*resp.* each object) is introduced by a unique concept and inherited by its sub-concepts (*resp.* by its super-concepts). In figures, objects and attributes are often represented only in their introducing concept for the sake of simplicity. Rather than using the whole concept lattice, which is often a large and complex structure (exponential number of concepts, considering objects and attributes, in the worst case), we use a sub-order called the

AOC-poset, which is restricted to the concepts that introduce at least one object or one attribute. The interested reader can find more information about our use of FCA in [3].

Relational Concept Analysis (RCA) [26] is an iterative version of FCA in which objects are classified not only according to attributes they share, but also according to relations between them (*cf.* Section 5.1). Other close approaches are [27, 28, 29].

In the RCA framework, 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 3). A RCF is iteratively processed to generate, at each step, a set of concept lattices. As a first step, 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 5).

To apply FCA and RCA, we use the Eclipse eRCA platform[b].

3.2. *Latent Semantic Indexing*

Information Retrieval (IR) refers to techniques that compute textual similarity between documents. Textual similarity is computed based on the occurrences of terms in documents [30]. When 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.

As proposed by [2], we use LSI to group together software artifacts that pertain to the implementation or the documentation of a similar, thus considered common, concept[c].

To do so, software artifacts are regarded as textual documents. Occurrences of terms are extracted from the documents in order to calculate similarities between them and then classify together similar documents (*cf.* Section 5.2).

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 overcome two issues of natural language processing: *synonymy* and *polysemy.*

The effectiveness of IR methods is usually measured by metrics including *recall*, *precision* and *F-measure* (*cf.* Equations 1, 2 and 3). In this work, 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

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

[c] To set our approach up, we developed our own LSI tool, available at `https://code.google.com/p/lirmmlsi/`

precision and recall, that gives a high value only when both recall and precision are high.

$$Recall = \frac{|\{relevant\ documents\} \bigcap \{retrieved\ documents\}|}{|\{relevant\ documents\}|} \quad (1)$$

$$Precision = \frac{|\{relevant\ documents\} \bigcap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|} \quad (2)$$

$$F - Measure = 2 \times \frac{Precision \cdot Recall}{Precision + Recall} \quad (3)$$

All measures have values in [0%, 100%]. When recall equals 100%, all relevant feature implementations are retrieved. However, some retrieved feature implementations may not be relevant. If precision equals 100%, all retrieved feature implementations are relevant. Nevertheless, all relevant feature implementations may not be retrieved. When F-Measure equals 100%, all relevant feature implementations are retrieved.

The interested reader can find more information about our use of LSI in [3].

3.3. *The Mobile Tourist Guide Example*

In this example, we consider four software variants of the Mobile Tourist Guide (MTG) application. These variants enable users to inquire about 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. Table 1 describes the sets of functionnalities (use-cases) implemented by the different MTG software variants. Figure 1 shows the corresponding use-case diagrams.

In this example, we can observe that the use-case diagrams of software variants show commonality and variability at use-case level (*i.e.*, functionalities). This prompts to extract feature documentation from the use-case diagrams of software variants. Table 2 shows the mined feature implementations from MTG software variants. In the examples, mined feature implementations are named using the same names as that of the corresponding use-case for the sake of clarity. But as mentioned before, mined feature names are not known beforehand. We only use the mined feature implementations composed of Source Code Elements (SCEs) and the use-case diagrams of software variants as inputs for the documentation process.

Table 1. The use-cases of MTG software variants.

	View map	Place marker on a map	View direction	Launch Google map	Show street view	Download map from Google	Show satellite view	Search for nearest attraction	Show next attraction	Retrieve data
Mobile Tourist Guide 1	×	×	×	×	×					
Mobile Tourist Guide 2	×	×	×	×	×	×				
Mobile Tourist Guide 3	×	×	×	×	×		×			
Mobile Tourist Guide 4	×	×	×	×	×			×	×	×

Product-by-use case matrix
(× use-case is in the product)

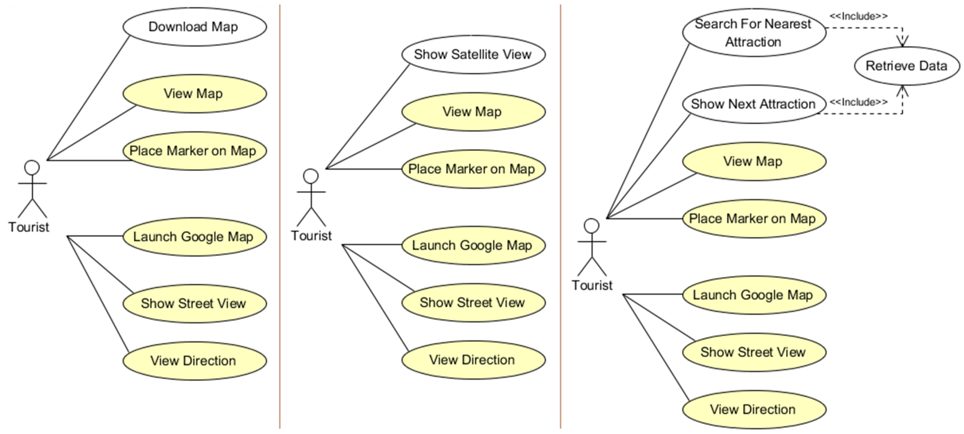


Fig. 1. The use-case diagrams of the MTG software variants.

4. The Feature Documentation Process

As previously mentioned, we aim at documenting mined feature implementations by using use-case diagrams that document a set of software variants. We rely on the same assumption as in the work of [15] stating that each use-case corresponds to a feature. The feature documentation process uses lexical similarity to identify which use-case best characterizes the name and description of each feature implementation. As performance and efficiency of the IR technique depend on the size of the search space, 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 5). We call each disjoint minimal cluster a *Hybrid Block* (HB). After reducing the search

Table 2. The mined feature implementations from MTG software variants.

	Feature Implementation_1: View map	Feature Implementation_2: Place marker on a map	Feature Implementation_3: View direction	Feature Implementation_4: Launch Google map	Feature Implementation_5: Show street view	Feature Implementation_6: Download map from Google	Feature Implementation_7: Show satellite view	Feature Implementation_8: Search for nearest attraction	Feature Implementation_9: Show next attraction	Feature Implementation_10: Retrieve data
Mobile Tourist Guide 1	×	×	×	×	×					
Mobile Tourist Guide 2	×	×	×	×	×	×				
Mobile Tourist Guide 3	×	×	×	×	×		×			
Mobile Tourist Guide 4	×	×	×	×	×			×	×	×

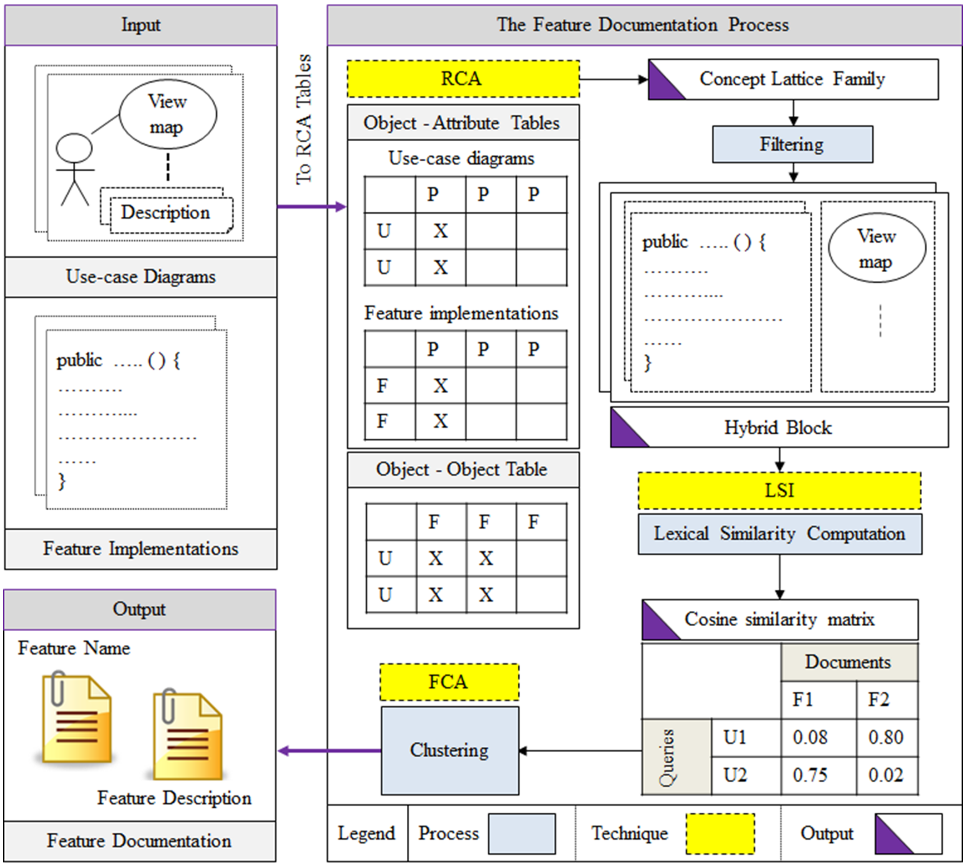
Product-by-feature implementation matrix
(× feature implementation is in the product)

space to a set of hybrid blocks, we measure textual similarity to identify, within each hybrid block, which use-case may provide a name and a 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 defined by its name and description. This information represents domain knowledge that is usually available as software documentation (*i.e.*, requirement model). In our work, the use-case description consists of a short paragraph in a natural language. For example, the *retrieve data* use-case of Figure 1 is described by the following paragraph, "*the tourist can retrieve information and a small image of the attraction using his/her mobile phone. In addition, the tourist can store the current view of the map in the mobile phone*".

Our approach provides a name and a description for each feature implementation based on a use-case name and description. Each use-case is mapped onto 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 2 shows an overview of our feature documentation process. The first step of this process identifies hybrid blocks based on RCA (*cf.* Section 5.1). In the second step, LSI is applied to determine similarity between use-cases and feature implementations (*cf.* Section 5.2). FCA is then used to build use-case clusters. Each



Use-case diagrams

	P	P	P
U	X		
U	X		

Feature implementations

	P	P	P
F	X		
F	X		

Object - Object Table

	F	F	F
U	X	X	
U	X	X	

Cosine similarity matrix

		Documents	
		F1	F2
Queries	U1	0.08	0.80
	U2	0.75	0.02

Fig. 2. The feature documentation process.

cluster identifies a name and a description for feature implementation (*cf.* Section 5.3).

5. Feature Documentation Step by Step

In this section, we describe the feature documentation process step by step. Feature name and description are identified through three steps as detailed in the following.

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

Mined feature implementations and use-cases are clustered into disjoint minimal clusters (*i.e.*, hybrid blocks) to apply LSI on reduced search spaces. RCA is used to build the clusters, based on the commonality and variability in software variants: use-cases and feature implementations that are common to all software variants;

use-cases and feature implementations that are shared by a set of software variants, but not all variants; use-cases and feature implementations that are held by a single variant (*cf.* Figure 3).

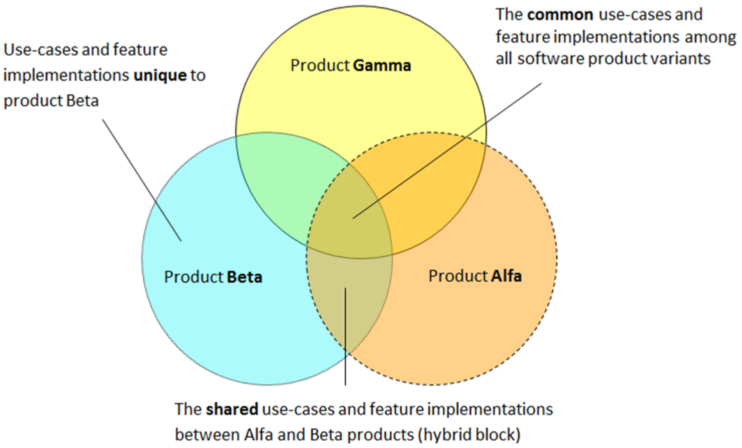

Use-cases and feature implementations **unique** to product Beta
Product **Gamma**
The **common** use-cases and feature implementations among all software product variants
Product **Beta**
Product **Alfa**
The **shared** use-cases and feature implementations between Alfa and Beta products (hybrid block)

Fig. 3. The common, shared and unique use-cases (*resp.* feature implementations) across software product variants.

An RCF is automatically generated from the use-case diagrams and the mined feature implementations associated with software variants[d]. The RCF used in our approach contains two formal contexts and one relational context, as illustrated in Table 3. The first formal context represents the *use-case diagrams*: objects are use-cases and attributes are software variants. The second formal context represents *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 variant as a feature implementation.

For the RCF in Table 3, the two lattices of the *Concept Lattice Family* (CLF) are represented in Figure 4 and a close-up view is represented in Figure 5. An example of a hybrid block is given in Figure 5 (the left dashed block), which gathers a set of use-cases (from the extent of *Concept_1* in the *Use_case_Diagrams* lattice) that always appear with a set of feature implementations (from the extent of *Concept_6* in the *Feature_Implementations* lattice). The relation between the two concepts is represented in *Concept_1* via the relational attribute *appears-with*:*Concept_6*. As shown in Figure 5, RCA enables to reduce the search space by exploiting commonality and variability across software variants. In our work, we filter the CLF from bottom to top to get a set of hybrid blocks[e].

[d] Source code : `https://code.google.com/p/rcafca/`
[e] Source code : `https://code.google.com/p/fecola/`

Table 3. The RCF for features documentation.

Use_case_Diagrams	MTG_1	MTG_2	MTG_3	MTG_4
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				×

Feature_Implementations	MTG_1	MTG_2	MTG_3	MTG_4
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				×

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								×	×	×

5.2. *Measuring the Lexical Similarity Between Use-cases and Feature Implementations via LSI*

Each hybrid block created during previous step consists of a set of use-cases and a set of feature implementations. Which use-cases characterize the name and description of each feature implementation needs to be identified. To do so, we use textual similarity between use-cases and feature implementations. This similarity measure is calculated using LSI. We consider that a use-case corresponding to a feature implementation should be lexically closer to this feature implementation than to others. Similarity between use-cases and feature implementations in the hybrid blocks is computed in three steps as detailed below.

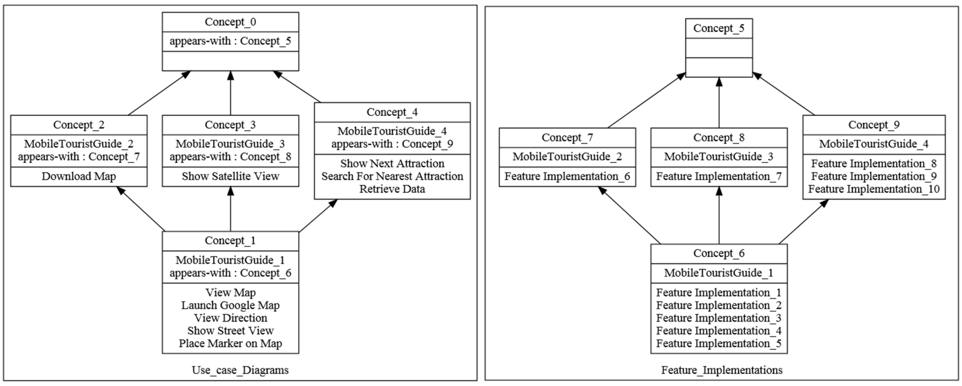


Fig. 4. The concept lattice family of relational context family in Table 3.

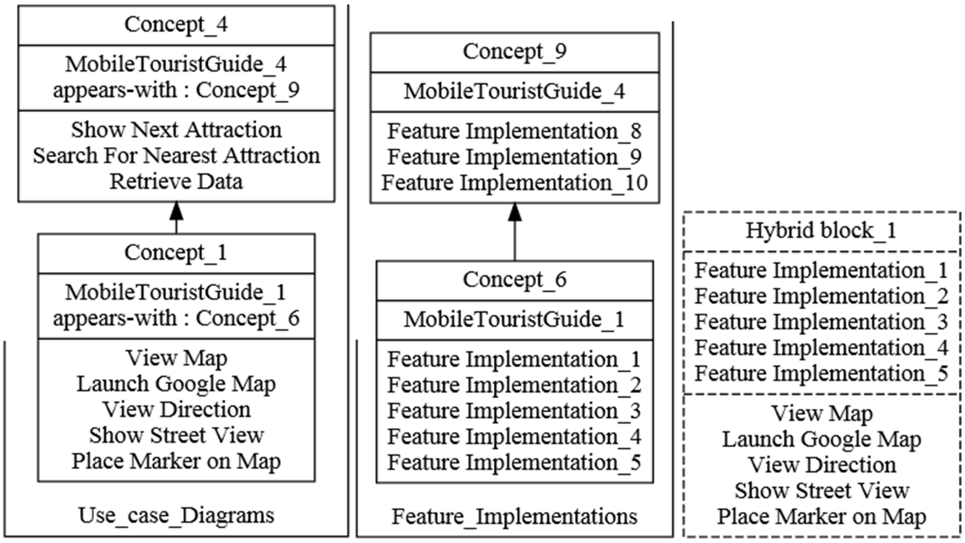
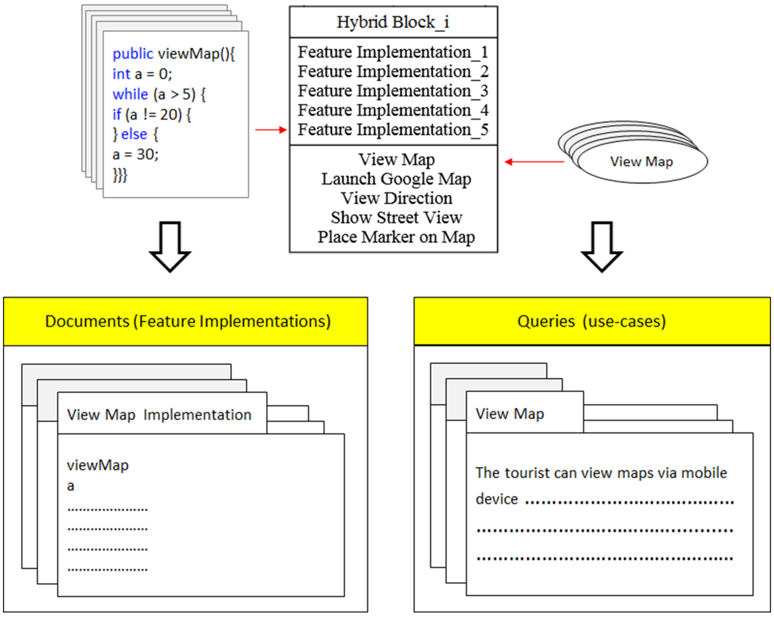


Fig. 5. Parts of the CLF deduced from Table 3.

5.2.1. *Building the LSI Corpus*

In order to apply LSI, we build a corpus that represents a collection of documents and queries (*cf.* Figure 6). 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 words extracted from the SCE names in the feature implementation as a result of splitting them using a tokenization scheme (Camel-case). Regardless of their location in the SCE names, we store all the words in the document. For example, in the SCE name *ManualTestWrapper* all words are



```
public viewMap(){
int a = 0;
while (a > 5) {
if (a != 20) {
} else {
a = 30;
}}}
```

Fig. 6. Constructing a raw corpus from hybrid block.

important: *manual*, *test* and *wrapper*. This process is applied to all feature implementations. Our approach creates a query for each use-case. This query contains the use-case name and its description.

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[f] to do part of the 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 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.

5.2.2. *Building the Term-document and the Term-query Matrices for each Hybrid Block*

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 oc-

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

currences of a term in a document, according to a specific weighting scheme. In our work, terms are weighted using the *TF-IDF* function (the most common weighting scheme) [2]. 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 in the term-query matrix refers to the weight of the i^{th} term in the j^{th} query.

Table 4. The term-document and the term-query matrices of *Concept_1* in Figure 5.

	Feature Implement._1	Feature Implement._2	Feature Implement._3	Feature Implement._4	Feature Implement._5
device	1	0	0	0	1
direction	0	0	0	6	0
google	1	0	0	0	0
launch	4	0	0	0	0
map	1	2	0	0	4
marker	0	6	0	0	0
mobile	1	0	0	0	1
place	0	3	0	0	0
show	0	0	2	0	0
street	0	0	5	0	0
tourist	1	1	1	1	1
view	0	0	1	2	5

The term-document matrix

	Launch Google Map	Place Marker on Map	Show Street View	View Direction	View Map
device	1	0	0	0	1
direction	0	0	0	8	0
google	3	0	0	0	0
launch	3	0	0	0	0
map	2	2	1	1	5
marker	0	3	0	0	0
mobile	1	0	0	0	1
place	0	3	0	0	0
show	0	0	3	0	0
street	0	0	5	0	0
tourist	1	1	1	1	1
view	0	0	1	3	5

The term-query matrix

In the term-document matrix (in left-hand side of Table 4), the *direction* term appears 6 times in the *Feature Implementation_4* document. In the term-query matrix (in right-hand side of Table 4), the *direction* term appears 8 times in the *view direction* query.

5.2.3. *Building the Cosine Similarity Matrix*

Similarity between documents in a corpus is measured by the cosine of the angle between their corresponding term vectors [31], as given by Equation 4, where d_q is a query vector, d_j is a document vector and $W_{i,q}$ and $W_{i,j}$ go over the weights of terms in the query and document respectively.

$$\textit{cosine similarity }(d_q, d_j) = \frac{\overrightarrow{d_q} \cdot \overrightarrow{d_j}}{|\overrightarrow{d_q}||\overrightarrow{d_j}|} = \frac{\sum\limits_{i=1}^{n} W_{i,q} * W_{i,j}}{\sqrt{\sum\limits_{i=1}^{n} W_{i,q}^2}\sqrt{\sum\limits_{i=1}^{n} W_{i,j}^2}} \tag{4}$$

Similarity between use-cases and feature implementations in each hybrid block is thus defined by a *cosine similarity matrix* which columns (documents) represent feature implementations and rows (queries) use-cases [3].

Table 5. The cosine similarity matrix of *Concept_1* in Figure 5.

	Feature Implementation_1	Feature Implementation_2	Feature Implementation_3	Feature Implementation_4	Feature Implementation_5
Launch Google Map	0.861933577	0.0137010	0	0	0.152407
Place Marker on Map	0.01114798	0.9480070	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	0.993111

Results are represented as a directed graph. Use-cases (*resp.* feature implementations) are represented as vertices and similarity links as edges. The degree of similarity appears as weights on the edges (*cf.* Figure 7). This graph is only used for visualization purposes.

5.3. *Identifying Feature Name via FCA*

Having the cosine similarity matrix, we use FCA to identify, in each hybrid block, which use-cases and feature implementations are related. To transform a (numerical) cosine similarity matrix into a (binary) formal context, we use a 0.70 threshold (after having tested many threshold values). This means that only pairs of use-cases and feature implementations having similarity greater than or equal to 0.70 are considered related. Table 6 shows the formal context obtained by transforming the cosine similarity matrix corresponding to the hybrid block of *Concept_1* from Figure 5.

For the *Concept_1* hybrid block of Figure 5 the number of term-topics of LSI is equal to 5. In the formal context associated with this hybrid block, the *"Launch Google Map"* use-case is linked to the *"Feature Implementation_1"* feature implementation because their similarity equals 0.86, which is greater than the threshold.

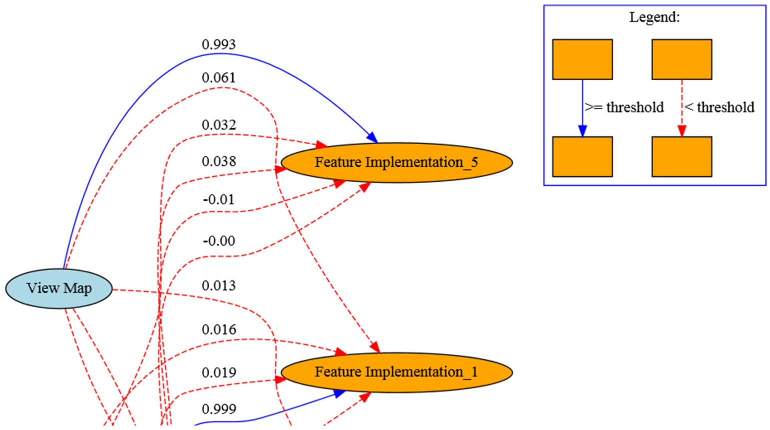


Fig. 7. The lexical similarity between use-cases and feature implementations as a directed graph.

Table 6. Formal context of *Concept_1* in Figure 5.

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

On the contrary, the *”View Direction”* use-case and the *”Feature Implementation_5”* feature implementation are not linked because their similarity equals 0.10, which is less than the threshold. The resulting AOC-poset is composed of concepts whose *extent* represents the use-case name(s) and *intent* represents the feature implementation(s). In this simple example, there is a one-to-one association between use case names and feature implementations, but in the general case, we may have several use case names associated with several feature implementations.

For the MTG example, the AOC-poset of Figure 8 shows five non comparable 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.

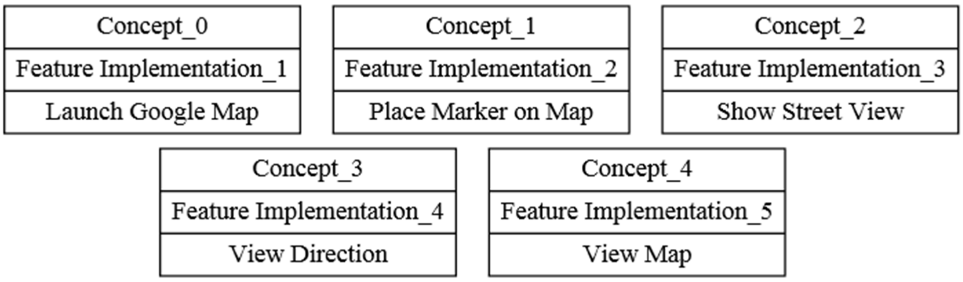


Fig. 8. The documented features from *Concept_1*.

6. Naming Feature Implementation Based on SCE Names

In our approach, we consider that use-case diagrams or other kinds of documentation (*i.e.*, design documents) are not always available. In case they are not, we use the source code of the mined features to automatically generate feature names and documentations.

We adapt the process proposed in [5]. Their work identifies component names based on *class names* in a single software. In our work, we extract a *name* for each *feature implementation* from the names given to its *SCEs*. We identify the name in three steps:

1. Extracting and tokenizing SCE names.
2. Weighting tokens.
3. Composing the feature name.

Our approach can be applied at any code granularity level (package, class, attribute, method, local variable, method invocation or attribute access).

- **Extracting and tokenizing SCE names.** At this step, the names of all the SCEs found in the feature implementation are extracted. Then, each SCE name is split into tokens, using a camel-case scheme. For example *getMinimumSupport* is split into *get*, *Minimum* and *Support*. This is a simple but common identifier splitting algorithm [32], conforming to programming best practices.
- **Weighting tokens.** At this step, a weight is assigned to each extracted token. A *high weight* (1.0) is given to the first word in a SCE name. A *medium weight* (0.7) is given to the second word in a SCE name. Finally a *small weight* (0.5) is given to the other words.
- **Composing the feature name.** In this step, a feature name is built using the highest weighted words.

The number of words used in the feature name is chosen by an *expert*. For example, the expert can select the top two words to construct the feature name. When many tokens have the same weight, all possible combinations are presented to the expert and he can choose the most relevant one. Table 7 shows an example of feature name proposals for the *show street view* feature implementation. In this example, the expert assigns a feature name based on the top three tokens. The assigned name for this feature implementation is eventually *StreetShowView*.

Table 7. SCE names, tokens, weight and highest weighted tokens for the *show street view* feature implementation.

	Token/Weight			
SCE Name	T1/ w = 1.0	T2/ w = 0.7	T3/ w = 0.5	T4/ w = 0.5
ShowStreetView	show	Street	View	
StreetPosition	Street	Position		
ChangeStreetSettings	Change	Street	Settings	
getStreetAddress	get	Street	Address	
setStreetAddress	set	Street	Address	
ShowNearestStreet	show	Nearest	Street	
ShowNextStreet	show	Next	Street	
retrieveStreetData	retrieve	Street	Data	
ShowStreet	show	Street		
updateStreetInfo	update	Street	Info	
ViewStreetMap	View	Street	Map	
ViewStreetPositionInfo	View	Street	Position	Info

Token	Total Weight	Top 3	Top 4
Show	4	×	×
Street	8	×	×
View	2.5	×	×
Position	1.2		×
Change	1		
Settings	1		
get	1		
Address	1		
set	1		
Nearest	0.7		
Next	0.7		
retrieve	1		
Data	0.5		
update	1		
Info	1		
Map	0.5		

7. Experimentation

7.1. *Experimental setup*

To validate our approach, we ran experiments on two Java open-source applications: Mobile media software variants (small systems) [33] and ArgoUML-SPL (large systems) [34]. We used 4 variants for Mobile media and 10 for ArgoUML. These two case studies interestingly implement variability at different levels: class and method levels. In addition, these case studies are well documented: their use-case diagrams and FMs are available for comparison and validation of our results[g]. Table 8 summarizes the obtained results.

7.2. *Results*

In these two case studies, we observe that the *recall* values are 100% for all features: this means that our approach efficiently associates uses cases with their implementations in the software. Precision values are in [50% - 100%]: similarity between a use-case and several features implementation may be high, when they pertain to the same application domain and thus use common vocabulary, leading to ill associations. F-Measure values are consequently in [66% - 100%]. In most cases, the

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

Table 8. 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_4	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%

contents of hybrid blocks are in the range of $[1-4]$ use-cases and feature implementations: this validates RCA as a valid technique for building small search spaces in order to efficiently compute lexical similarity. Lexical similarity also proves to be a suitable tool, as shown by high recall, which confirms that a common vocabulary is used in use-case descriptions and feature implementations.

In our work, we cannot use a fixed number of topics for LSI because hybrid blocks (clusters) have different sizes. The column (*k-Topics*) in Table 8 represents *the number of term-topics.*

All feature names produced by our approach are presented in the column (*Feature Name*) of Table 8, as built from names of use-cases. For example, in the FM of Mobile media [33] 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*".

7.3. *Threats to validity*

There is a limit to the use of FCA as a clustering technique. Cosine similarity matrices are transformed into formal (binary) contexts thanks to a fixed threshold.

So if similarity between the query and the document is greater than or equals 0.70 the two documents are considered similar. By contrast, if similarity is less than the threshold (*i.e.*, 0.69) the two documents are considered dissimilar. This sharp threshold effect may affect the quality of the result, since a similarity value of 0.99 (*resp.* 0.69) is treated as a similarity value 0.70 (*resp.* 0). Adaptive and fuzzy threshold schemes should be studied to improve precision without impacting the high recall of our approach.

In our approach we consider that each use-case corresponds to a functional feature. However, several use-cases may be implemented by a single feature. In this case all these use-cases should be considered as relevant documentation for this feature. Our approach should be improved with other techniques to extract a unique name and a compound description.

Naming a feature using the names of the SCEs in its implementation is not always reliable. In our approach, we rely on the top word frequencies to compose the proposed name. However, top words may be not relevant to depict the function of the feature. The weighting scheme should take into account the different roles and importance of SCEs in the implementation in order to select the most relevant words, if they are not the most frequent ones.

8. Conclusion and Perspectives

In this paper, we propose an approach for documenting automatically a set of feature implementations mined from a set of software variants. We exploit commonalities and variabilities between software variants at feature implementation and use-case levels in order to apply IR methods in an efficient way. We have implemented our approach and evaluated its results on two case studies. The good results (high recall) validates the main principles of our approach. Regarding future work, we would like to improve the clustering into hybrid blocks using other techniques. We would also like to improve the precision and relevance of our results thanks to adaptive similarity thresholds and semantic weighting schemes. We also plan to extract relations between the mined and documented features and automatically build a FM in order to support a complete reverse engineering process from source code to a SPL.

Acknowledgements

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

References

[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, ser. The SEI series in software engineering. Addison Wesley Professional, 2002.

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

[3] 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 *Proceedings of The 25th International Conference on Software Engineering and Knowledge Engineering*, 2013, pp. 244–249.

[4] H. A. Müller, S. R. Tilley, and K. Wong, "Understanding software systems using reverse engineering technology perspectives from the rigi project," in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, ser. CASCON '93. IBM Press, 1993, pp. 217–226.

[5] S. Kebir, A.-D. Seriai, S. Chardigny, and A. Chaoui, "Quality-centric approach for software component identification from object-oriented code," in *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, ser. WICSA-ECSA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 181–190.

[6] 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.

[7] 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.

[8] 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*, D. Beyer, A. van Deursen, and M. W. Godfrey, Eds., 2012, pp. 193–202.

[9] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 223–226.

[10] 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 *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ser. ICPC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 4–13.

[11] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52.

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

[13] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 125–135.

[14] D. Diaz, G. Bavota, A. Marcus, R. Oliveto, S. Takahashi, and A. D. Lucia, "Using code ownership to improve ir-based traceability link recovery," in *ICPC*, 2013, pp. 123–132.

[15] 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.

[16] Y. Yang, X. Peng, and W. Zhao, "Domain feature model recovery from multiple applications using data access semantics and Formal Concept Analysis," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 215–224.

[17] P. Paškevičius, R. Damaševičius, E. karčiauskas, and R. Marcinkevičius, "Automatic extraction of features and generation of feature models from Java programs," *Information Technology and Control*, pp. 376 – 384, 2012.

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

[19] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Feature location in a collection of software product variants using Formal Concept Analysis," in *ICSR*, ser. Lecture Notes in Computer Science, J. M. Favaro and M. Morisio, Eds., vol. 7925. Springer, 2013, pp. 302–307.

[20] R. Al-Msie'deen, A.-D. 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*. IEEE, 2013, pp. 586–593.

[21] 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/SIGSOFT FSE*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 290–300.

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

[23] T. Tilley, R. Cole, P. Becker, and P. W. Eklund, "A survey of formal concept analysis support for software engineering activities," in *Formal Concept Analysis*, ser. Lecture Notes in Computer Science, B. Ganter, G. Stumme, and R. Wille, Eds., vol. 3626. Springer, 2005, pp. 250–271.

[24] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux, "Formal Concept Analysis enhances fault localization in software," in *ICFCA*, ser. Lecture Notes in Computer Science, R. Medina and S. A. Obiedkov, Eds., vol. 4933. Springer, 2008, pp. 273–288.

[25] M. U. Bhatti, N. Anquetil, M. Huchard, and S. Ducasse, "A catalog of patterns for concept lattice interpretation in software reengineering," in *SEKE*. Knowledge Systems Institute Graduate School, 2012, pp. 118–123.

[26] 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.

[27] S. Prediger and R. Wille, "The lattice of concept graphs of a relationally scaled context," in *ICCS*, ser. Lecture Notes in Computer Science, W. M. Tepfenhart and W. R. Cyre, Eds., vol. 1640. Springer, 1999, pp. 401–414.

[28] S. Ferré, O. Ridoux, and B. Sigonneau, "Arbitrary relations in Formal Concept Analysis and logical information systems," in *ICCS*, ser. Lecture Notes in Computer Science, F. Dau, M.-L. Mugnier, and G. Stumme, Eds., vol. 3596. Springer, 2005, pp. 166–180.

[29] F. Baader and F. Distel, "A finite basis for the set of EL-implications holding in a finite model," in *ICFCA*, ser. Lecture Notes in Computer Science, R. Medina and S. A. Obiedkov, Eds., vol. 4933. Springer, 2008, pp. 46–61.

[30] D. Grossman and O. Frieder, *Information Retrieval: Algorithms and Heuristics*, ser. Kluwer international series in engineering and computer science. Springer, 2004.

[31] M. Berry and M. Browne, *Understanding Search Engines: Mathematical Modeling and Text Retrieval*, ser. ITPro collection. Society for Industrial and Applied Mathematics, 1999.

[32] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ser. ICPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 11–20.

[33] 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.

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

[35] R. Medina and S. A. Obiedkov, Eds., *Formal Concept Analysis, 6th International Conference, ICFCA 2008, Montreal, Canada, February 25-28, 2008, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4933. Springer, 2008.