Automatic Documentation of [Mined] Feature Implementations from Source Code Elements and Use-Case Diagrams with the REVPLINE Approach

Ra’Fat Ahmad Al-Msie’Deen, Marianne Huchard, Abdelhak-Djamel Seriai, Christelle Urtado, Sylvain Vauttier

To cite this version:


HAL Id: lirmm-01147898

https://hal-lirmm.cnrs.fr/lirmm-01147898

Submitted on 1 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Companies often develop a set of software variants that share some features and differ in others 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 to re-engineer them into an 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\(^\text{a}\)). 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. Sec. 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, Sec. 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 documenting 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 documenting the source code of a single software, finding traceability links between source code and documentation, and documenting 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 [23–25]. 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. Sec. 5.1). Other close approaches are [27–29].

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. Fig. 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. Sec. 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. Eqs. (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

\(^b\)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/
number of retrieved feature implementations. F-Measure defines a trade-off between precision and recall, that gives a high value only when both recall and precision are high.

\[
Recall = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|} \quad (1)
\]

\[
Precision = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|} \quad (2)
\]

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

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 functionalities (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.
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 Fig. 5). We call

---

Table 1. The use cases of MTG software variants.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Mobile Tourist Guide 1</th>
<th>Mobile Tourist Guide 2</th>
<th>Mobile Tourist Guide 3</th>
<th>Mobile Tourist Guide 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>View map</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Place marker on a map</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>View direction</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Launch Google map</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Show street view</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Show download map from Google</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Show satellite view</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Search for nearest attraction</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Show next attraction</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

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

---

Fig. 1. The use-case diagrams of the MTG software variants.
each disjoint minimal cluster a Hybrid Block (HB). After reducing the search 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 Fig. 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.

<table>
<thead>
<tr>
<th>Feature Implementation</th>
<th>View map</th>
<th>Place marker on a map</th>
<th>View direction</th>
<th>Launch Google map</th>
<th>Show street view</th>
<th>Download map from Google</th>
<th>Show satellite view</th>
<th>Search for nearest attraction</th>
<th>Show next attraction</th>
<th>Retrieve data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile Tourist Guide 1</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Mobile Tourist Guide 2</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Mobile Tourist Guide 3</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Mobile Tourist Guide 4</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Product-by-feature implementation matrix (× feature implementation is in the product)
Figure 2 shows an overview of our feature documentation process. The first step of this process identifies hybrid blocks based on RCA (cf. Sec. 5.1). In the second step, LSI is applied to determine similarity between use cases and feature implementations (cf. Sec. 5.2). FCA is then used to build use-case clusters. Each cluster identifies a name and a description for feature implementation (cf. Sec. 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. Fig. 3).

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 Fig. 4 and a close-up view is represented in Fig. 5. An example of a hybrid block is given in Fig. 5 (the left dashed block), which gathers a set of use cases (from the extent of Concept1 in the Use_case_Diagrams lattice) that always appear with a set of feature implementations (from the extent of Concept6 in the Feature_Implementations lattice). The relation between the two concepts is represented in Concept1 via the relational attribute appears-with:Concept6. As shown in

\(^d\)Source code: https://code.google.com/p/rcafca/
### Table 3. The RCF for features documentation.

<table>
<thead>
<tr>
<th>Use_case_diagrams</th>
<th>MTG-1</th>
<th>MTG-2</th>
<th>MTG-3</th>
<th>MTG-4</th>
<th>Feature_Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>View Map</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Feature Implementation_1</td>
</tr>
<tr>
<td>Launch Google Map</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Feature Implementation_2</td>
</tr>
<tr>
<td>View Direction</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Feature Implementation_3</td>
</tr>
<tr>
<td>Show Street View</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Feature Implementation_4</td>
</tr>
<tr>
<td>Place Marker on Map</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Feature Implementation_5</td>
</tr>
<tr>
<td>Download Map</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>Feature Implementation_6</td>
</tr>
<tr>
<td>Show Satellite View</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>Feature Implementation_7</td>
</tr>
<tr>
<td>Show Next Attraction</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>Feature Implementation_8</td>
</tr>
<tr>
<td>Search for nearest attraction</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>Feature Implementation_9</td>
</tr>
<tr>
<td>Retrieve Data</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>Feature Implementation_10</td>
</tr>
</tbody>
</table>

Relational context: appears-with

<table>
<thead>
<tr>
<th>Feature_Implementations</th>
<th>MTG-1</th>
<th>MTG-2</th>
<th>MTG-3</th>
<th>MTG-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>View Map</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Launch Google Map</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>View Direction</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Show Street View</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Place Marker on Map</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Download Map</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Show Satellite View</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Show Next Attraction</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Search for nearest attraction</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Retrieve Data</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Fig. 4. The concept lattice family of relational context family in Table 3.
Fig. 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.

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.

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. Fig. 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.

---

Source code: https://code.google.com/p/fecola/

---
(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 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\(^4\) to do part of the pre-processing (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.

\(^4\)http://wordnet.princeton.edu/
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 occurrences 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.

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 Eq. (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

<table>
<thead>
<tr>
<th>Concept_1 in Figure 5.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Implement.</td>
</tr>
<tr>
<td>device</td>
</tr>
<tr>
<td>direction</td>
</tr>
<tr>
<td>google</td>
</tr>
<tr>
<td>launch</td>
</tr>
<tr>
<td>map</td>
</tr>
<tr>
<td>marker</td>
</tr>
<tr>
<td>mobile</td>
</tr>
<tr>
<td>place</td>
</tr>
<tr>
<td>show</td>
</tr>
<tr>
<td>street</td>
</tr>
<tr>
<td>tourist</td>
</tr>
<tr>
<td>view</td>
</tr>
</tbody>
</table>

The term-document matrix  The term-query matrix
in the query and document respectively.

\[
\text{cosine similarity} \ (d_q, d_j) = \frac{d_q \cdot d_j}{\|d_q\| \|d_j\|} = \frac{\sum_{i=1}^{n} W_{i,q} \ast W_{i,j}}{\sqrt{\sum_{i=1}^{n} W_{i,q}^2} \sqrt{\sum_{i=1}^{n} W_{i,j}^2}}
\]

(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].

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. Fig. 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 Fig. 5.

Fig. 7. The lexical similarity between use-cases and feature implementations as a directed graph.
For the Concept_1 hybrid block of Fig. 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. 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 Fig. 8 shows five non comparable concepts (that correspond to five distinct features) mined from a single hybrid block (Concept_1 from Fig. 5). The same feature documentation process is used for each hybrid block.

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>
<thead>
<tr>
<th>Token/weight</th>
<th>SCE Name</th>
<th>Total weight</th>
<th>Top 3</th>
<th>Top 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1/w</td>
<td>ShowStreetView</td>
<td>1.0</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>T2/w</td>
<td>Street</td>
<td>0.7</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>T3/w</td>
<td>View</td>
<td>0.5</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>T4/w</td>
<td>Position</td>
<td>0.5</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 7. SCE names, tokens, weight and highest weighted tokens for the *show street view* feature implementation.
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\(^8\). 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

\(^8\)Case studies and code: http://www.lirmm.fr/CaseStudy

<table>
<thead>
<tr>
<th>#</th>
<th>Feature name</th>
<th>Hybrid block</th>
<th>k-Topics</th>
<th>Recall (%)</th>
<th>Precision (%)</th>
<th>F-Measure (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Delete Album</td>
<td>HB_.1</td>
<td>4</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>Delete Photo</td>
<td>HB_.1</td>
<td>4</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
<tr>
<td>3</td>
<td>Add Album</td>
<td>HB_.1</td>
<td>4</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Add Photo</td>
<td>HB_.1</td>
<td>4</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
<tr>
<td>5</td>
<td>Exception handling</td>
<td>HB_.2</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>Count Photo</td>
<td>HB_.3</td>
<td>3</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
<tr>
<td>7</td>
<td>View Sorted Photos</td>
<td>HB_.3</td>
<td>3</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
<tr>
<td>8</td>
<td>Edit Label</td>
<td>HB_.3</td>
<td>3</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Set Favourites</td>
<td>HB_.4</td>
<td>2</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
<tr>
<td>10</td>
<td>View Favourites</td>
<td>HB_.4</td>
<td>2</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
</tbody>
</table>

*Table 8. Features documented from case studies.*

---

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\(^8\). 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

\(^8\)Case studies and code: http://www.lirmm.fr/CaseStudy

<table>
<thead>
<tr>
<th>#</th>
<th>Feature name</th>
<th>Hybrid block</th>
<th>k-Topics</th>
<th>Recall (%)</th>
<th>Precision (%)</th>
<th>F-Measure (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Delete Album</td>
<td>HB_.1</td>
<td>4</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>Delete Photo</td>
<td>HB_.1</td>
<td>4</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
<tr>
<td>3</td>
<td>Add Album</td>
<td>HB_.1</td>
<td>4</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Add Photo</td>
<td>HB_.1</td>
<td>4</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
<tr>
<td>5</td>
<td>Exception handling</td>
<td>HB_.2</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>Count Photo</td>
<td>HB_.3</td>
<td>3</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
<tr>
<td>7</td>
<td>View Sorted Photos</td>
<td>HB_.3</td>
<td>3</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
<tr>
<td>8</td>
<td>Edit Label</td>
<td>HB_.3</td>
<td>3</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Set Favourites</td>
<td>HB_.4</td>
<td>2</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
<tr>
<td>10</td>
<td>View Favourites</td>
<td>HB_.4</td>
<td>2</td>
<td>100</td>
<td>50</td>
<td>66</td>
</tr>
</tbody>
</table>

*Table 8. Features documented from case studies.*

---
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 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 \((k\text{-Topics})\) 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) validate 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

