Mining Features from the Object-Oriented Source Code of a Collection of Software Variants Using Formal Concept Analysis and Latent Semantic Indexing

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

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

Abstract

Companies often develop a set of software variants that share some features and differ in other ones to meet specific requirements. To exploit existing software variants and build a software product line (SPL), a feature model of this SPL must be built as a first step. To do so, it is necessary to mine optional and mandatory features from the source code of the software variants. Thus, we propose, in this paper, a new approach to mine features from the object-oriented source code of a set of software variants based on Formal Concept Analysis and Latent Semantic Indexing. To validate our approach, we applied it on ArgoUML and Mobile Media case studies. The results of this evaluation validate the relevance and the performance of our proposal as most of the features were correctly identified.

***Keywords**: Software product line engineering, software variants, feature mining, FCA, LSI.*

1 Introduction

A software product line (SPL) is "a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way" [1]. A SPL is usually characterized by two sets of features: the features that are shared by all products in the family, which represent the *SPL's commonalities*, and the features that are shared by some, but not all, products in the family, which represent the *SPL's variability*. SPLs are usually described with a *de-facto* standard formalism called*feature model* [1]. A feature model defines all the valid feature configurations. In common software development processes software product variants often evolve from an initial product developed for and successfully used by the first customer. Mobile Media Systems [2] is an example of such a product evolution. These product variants usually share some common features but they are also different from one another due to subsequent customization to meet the specific requirements of different customers [3]. When variants become numerous, switching to a rigorous software product line engineering (SPLE) process is a solution to tame the increasing complexity of all the engineering tasks. To switch to SPLE starting from a collection of existing variants, the first step is to mine a feature model that describes the SPL. This implies to identify the software family's common and variable features. Manual reverse engineering of a feature model for software variants is time-consuming, error-prone, and requires substantial efforts [4]. Assisting this process would be of great help. This paper proposes an approach to mine features from a collection of software product variants in order to define the feature model of the software family[1]. Our approach is based on the identification of the implementation of these features among object-oriented (OO) elements of the source code. These OO elements constitute the initial search space. We use Formal Concept Analysis (FCA) to reduce this search space by first separating common and variable elements and, secondly, dividing the set of variable elements in subgroups. We further use Latent Semantic Indexing (LSI) to define a similarity measure that enables to identify subgroups of elements that characterize the implementation of each possible feature. Our approach is detailed in the remainder of this paper as follows. Section 2 sketches out a background of the used classification techniques. Section 3 presents the principles and main concepts of our approach. Section 4 details the feature mining process step by step. Section 5 describes the experiments that were con-

[1]This work has been funded by grant ANR 2010 BLAN 021902

ducted to validate our proposal. Section 6 discusses related work. Section 7 concludes and provides perspectives for this work.

2 Background: Techniques Used for Classification

This section presents a quick overview of the two techniques – Formal Concept Analysis (FCA) and Latent Semantic Indexing (LSI) – we plan to combine to classify information on software variants in order to extract features from their source code.

2.1 Formal Concept Analysis

Galois lattices and concept lattices [5] are core structures of the Formal Concept Analysis framework (FCA), which enables to extract an ordered set of concepts from a dataset, called a formal context, composed of objects described by attributes. A formal context is a triple $K = (O, A, R)$ where O and A are sets (objects and attributes respectively) and R is a binary relation, *i.e.*, $R \subseteq O \times A$. Several examples of formal contexts are provided in the remaining of this paper. A formal concept is a pair (E, I) composed of an object set $E \subseteq O$ and their shared attribute set $I \subseteq A$. $E = \{o \in O | \forall a \in I, (o, a) \in R\}$ is the extent of the concept, while $I = \{a \in A | \forall o \in E, (o, a) \in R\}$ is the intent of the concept. Given a formal context $K = (O, A, R)$ and two formal concepts $C_1 = (E_1, I_1)$ and $C_2 = (E_2, I_2)$ of K, the concept specialization order $\leq_s$ is defined by $C_1 \leq_s C_2$ if and only if $E_1 \subseteq E_2$ (and equivalently $I_2 \subseteq I_1$). C_1 is called a sub-concept of C_2. C_2 is called a super-concept of C_1. Let $\mathcal{C}_K$ be the set of all concepts of a formal context K. This set of concepts provided with the specialization order ($\mathcal{C}_K, \leq_s$) has a lattice structure, and is called the concept lattice associated with K. In our approach, we will consider the AOC-poset (for Attribute-Object-Concept poset), which is the sub-order of ($\mathcal{C}_K$, $\leq_s$) restricted to object-concepts and attribute-concepts (AOC-poset ignore concepts empty of declared objects and attributes). AOC-posets scale much better than lattices. Several figures in the remaining of this paper show AOC-posets. For readability's sake, in these diagrams, extents and intents are presented in a simplified form, removing top-down inherited attributes and bottom-up included objects.

2.2 Latent Semantic Indexing

Several IR methods exist such as Vector Space Method (VSM) and Latent Semantic Indexing (LSI) [6]. Both methods assume that software artifacts can be regarded as textual documents. Occurrences of terms are extracted from the documents in order to calculate similarities between them and then to classify together a set of similar documents as related to a common concept. LSI is an advanced IR method. The heart of LSI is singular value decomposition technique (SVD). This technique is used to mitigate noise introduced by stop words like (the, an, above) and to overcome two classic problems arising in natural languages processing: synonymy and polysemy [6]. The effectiveness of IR methods is measured by their recall, precision and F-Measure. For a given query, recall is the percentage of correctly retrieved results to the total number of relevant results, while precision is the percentage of correctly retrieved results to the total number of retrieved results. F-Measure defines a tradeoff between precision and recall with a high value only in cases where both recall and precision are high. All measures have values between [0, 1]. If recall equals 1, all relevant results are retrieved. However, some retrieved results might not be relevant. If precision equals 1, all retrieved results are relevant. However, relevant results might not be retrieved [6]. If F-Measure equals 1, all relevant results are retrieved and only relevant results are retrieved.

3 Approach Basics

This section presents the main concepts and hypotheses used in our approach for mining features from source code. It also shortly describes the example that illustrates the remaining of the paper.

3.1 Goal and Core Assumptions

The general objective of our work is to mine the feature model of a collection of software product variants based on the static analysis of their source code. Mining common and variable features is a first necessary step towards this objective. We consider that "a feature is a prominent or distinctive and user visible aspect, quality, or characteristic of a software system or systems" [7]. Our work focuses on the mining of functional features. Functional features express the behaviour or the way users may interact with a product. As there are several ways to implement features [8], we consider software systems in which functional features are implemented at the programming language level (*i.e.*, source code). We also restrict to OO software. Thus, features are implemented using object-oriented building elements (OBEs) such as packages, classes, attributes, methods or method body elements (local variable, attribute access, method invocation). We consider that a feature corresponds to one and only one set of OBEs. This means that a feature always has the same implementation in all products where it is present. We also consider that feature implementations may overlap: a given OBE can be shared between several features' implementation.

3.2 Features versus Object-oriented Building Elements: the Mapping Model

Mining a feature from the source code of variants amounts to identify group of OBEs that constitutes its implementation. This group of OBEs must either be present in all variants (case of a common feature) or in some but not all variants (case of an optional feature). Thus, the initial search space for the feature mining process is composed of all the OBEs in the existing product variants. For a source code containing *n* OBEs, the initial search space is the powerset of *n* deprived of the empty set. As the number of OBEs is high, mining features entails to reduce this search space. Several strategies can be combined to do so:

- separate the OBE set in two subsets, the common features set – also called *common block* (CB) – and the optional features set, on which the same search process will have to be performed. Indeed, as optional (*resp.* common) features appear in some but not all (*resp.* all) variants, they are implemented by OBEs that appear in some but not in all (*resp.* all) variants.
- separate the optional feature set into small subsets that each contains OBEs shared by groups of two or more variants or OBEs that are hold uniquely by a given variant. Each of these subsets is called a *block of variation* (BV). BVs can then be considered as smaller search spaces that each corresponds to the implementation of one or more features.
- identify common atomic blocks (CAB) amongst common block based on the expected lexical similarity between the OBEs that implement a given feature. A CB is thus composed of several CABs.
- identify atomic blocks of variation (ABV) inside of each BV based on the expected lexical similarity between the OBEs that implement a given feature. A BV is thus composed of several ABVs.

All the concepts we defined for mining features are illustrated in the OBE to feature mapping model of Figure 1.

3.3 An Illustrative Example

As an illustrative example, we consider three text editor software variants. *Editor_1* supports core text editing features: *open*, *close* and *print* a file. *Editor_2* has the core text editing features and a new *select_all* feature. *Editor_3* supports *copy* and *paste* features, together with the core ones. In this example, the eventually mined features are presented to better explain some parts of our work. However, we only use the source code of software variants as input of the mining process and thus do not know features in advance.

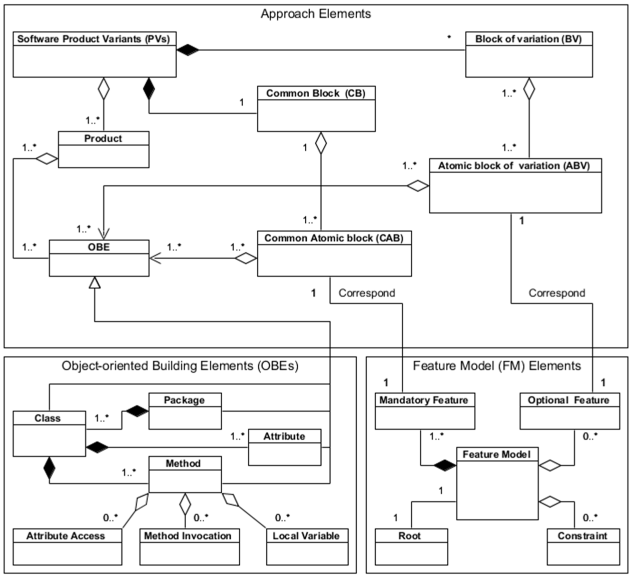

Figure 1: OBE to Feature Mapping Model.

4 The Feature Mining Process

The mapping model between OBEs and features defines associations between these features and the corresponding OBEs. To determine instances of this model, we describe our feature mining process. This process takes the variants' source code as its input. The first step of this process aims at identifying BVs and the CB based on FCA (*cf.* Section 4.1). The second step explores the AOC-poset of BVs to define an order to search for atomic blocks of variation (*cf.* Section 4.2.1). In the third step, we rely on LSI to determine the similarity between OBEs (*cf.* Section 4.2.2). This similarity measure is used to identify atomic blocks based on OBE clusters (*cf.* Section 4.2.3). Figure 2 shows our feature mining process.

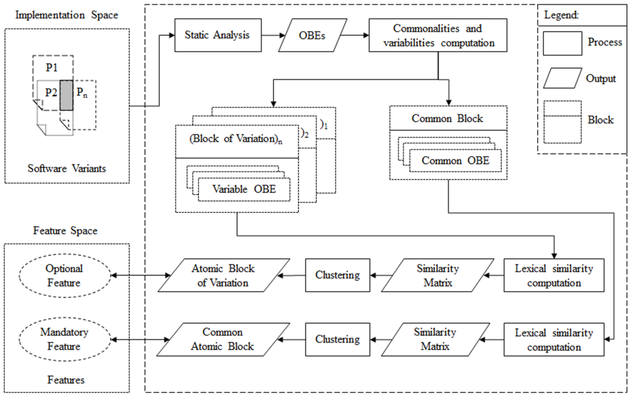

Figure 2: The Feature Mining Process.

4.1 Identifying the Common Block and Blocks of Variation

The first step of our feature mining process is the identification of the common OBE block and of OBE blocks of variation. The role of these blocks is to be sub-search spaces for mining sets of OBEs that implement features.

The technique used to identify the CB and BVs relies on FCA. First, a formal context, where objects are product variants and attributes are OBEs (*cf.* Table 1), is defined. The corresponding AOC-poset is then calculated. The intent of each concept represents OBEs common to two or more products. As concepts of AOC-posets are ordered, the intent of the most general (*i.e.*, top) concept gathers OBEs that are common to all products. They constitute the CB. The intents of all remaining concepts are BVs. They gather sets of OBEs common to a subset of products and correspond to the implementation of one or more features. The extent of each of these concepts is the set of products having these OBEs in common (*cf.* Figure 3).

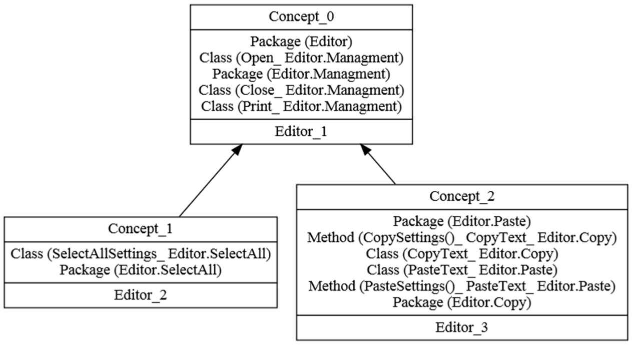
Concept_0
Package (Editor)
Class (Open_ Editor.Managment)
Package (Editor.Managment)
Class (Close_ Editor.Managment)
Class (Print_ Editor.Managment)
Editor_1
Concept_1
Class (SelectAllSettings_ Editor.SelectAll)
Package (Editor.SelectAll)
Editor_2
Concept_2
Package (Editor.Paste)
Method (CopySettings()_ CopyText_ Editor.Copy)
Class (CopyText_ Editor.Copy)
Class (PasteText_ Editor.Paste)
Method (PasteSettings()_ PasteText_ Editor.Paste)
Package (Editor.Copy)
Editor_3

Figure 3: The AOC-poset for the formal context of Table 1.

4.2 Identifying Atomic Blocks

The CB and BVs might each implement several features. Identifying the OBEs that characterize a feature's implementation thus consists in separating OBEs from the CB or from each of the BVs in smaller sets called atomic blocks. Atomic blocks are identified based on the calculation of the similarity between OBEs from the CB or a BV. These similarities result from applying LSI. Atomic blocks are clusters of the most similar OBEs built with FCA as detailed in the following.

Table 1: The formal context for the Text Editor Variants.

	Package(Editor.Managment)	Class (Close_Editor.Managment)	Class (Open_Editor.Managment)	Class (Print_Editor.Managment)	Package(Editor.Copy)	Class (CopyText_Editor.Copy)	Method (CopySettings_CopyText)	Package(Editor.SelectAll)	Class (SelectAllSettings_SelectAll)	Package(Editor.Paste)	Class (PasteText_Editor.Paste)	Method (PasteSettings_PasteText)
Editor_1	×	×	×	×								
Editor_2	×	×	×	×				×	×			
Editor_3	×	×	×	×	×	×	×			×	×	×

4.2.1 Exploring the BV's AOC-poset to Identify Atomic Blocks of Variation

As concepts of the AOC-poset are ordered, the search for atomic blocks of variation (ABVs) can be optimized if exploring the AOC-poset from the smallest (bottom) to the highest (top) block. Results (ABVs) obtained for a concept are used in the exploration of next (*i.e.*, upper) concepts: if a group of OBEs is identified as an ABV, this group is considered as such when exploring the following BV. For Common Atomic Blocks (CAB), there is no such need to explore the AOC-poset as there is a unique CB.

4.2.2 Measuring OBEs' Similarity Based on LSI

OBEs of BVs or of the CB respectively characterize the implementation of optional and mandatory features. We base the identification of subsets of OBEs, which each constitutes a feature, on the measurement of lexical similarity between these OBEs. This similarity measure is calculated using LSI. We rely on the fact that OBEs involved in implementing a functional feature are lexically closer to one another than to the rest of OBEs. To compute similarity between each pair of OBEs in the CB and BVs, we proceed in three steps: building the LSI corpus, building the term-document matrix and the term-query matrix for each BV and for the CB and, at last, building the cosine similarity matrix.

Building the LSI corpus. In order to apply LSI, we build a corpus that represents a collection of documents and queries. In our case, each OBE in the block represents both a document and a query. To be processed, the document and query must be normalized (*e.g.*, all capitals turned into lower case letters, articles, punctuation marks or numbers removed). The normalized document generated by analyzing the source code of an OBE is splitted into terms and, at last, word stemming is performed.

Building the term-document and the term-query matrices for each block. All blocks (the CB and all BVs) are considered and applied the same process. The term-document matrix is of size $m \times n$ where m is the number of terms used in a normalized document corresponding to an OBE and n the number of OBEs in a block. In the same way, a term-query matrix is of size $m \times j$ where m is the number of terms and j the number of OBEs. Each column in the term-query matrix represents a vector of OBEs. Terms for both matrices are the same because they are extracted from the same block.

Building the similarity matrix. Similarity between OBEs in each BV or in the CB is described by a cosine similarity matrix whose columns and rows both represent vectors of OBEs: documents as columns and queries as rows. Similarity is computed as a cosine similarity given by Equation 1, where Q_i is a query vector, D_j is a document vector and W_i and W_j range over weights of query and document vectors, respectively.

$$CosineSimilarity(Q_i, D_j) = \frac{\sum_{i=1}^{n} W_i * W_j}{\sqrt{\sum_{i=1}^{n} W_i^2 \sum_{j=1}^{n} W_j^2}} \quad (1)$$

4.2.3 Identifying Atomic Blocks Using FCA

We then use FCA to identify, from each block of OBEs, which elements are similar. To transform the (numerical) similarity matrices of previous step into (binary) formal contexts, we use a threshold. 0.70 is the chosen threshold value (a widely used threshold for cosine similarity [6]) meaning that only pairs of OBEs having a calculated similarity greater than or equal to 0.70 are considered similar. Table 2 shows the formal context obtained by transforming the similarity matrix corresponding to the BV of *Concept_2* from Figure 3. As an example, in the formal context of this table, the OBE *"Method PasteSetting PasteText"* is linked to the OBE *"Class PasteText Paste"* because their similarity equals 0.99, which is greater than the threshold. However, the OBE *"Method CopySettings CopyText"* and the OBE *"Class PasteText Paste"* are not linked because their similarity equals 0.18, which is less than the threshold. The resulting AOC-poset is composed of concepts the extent and intent[2] of which group similar OBEs.

Table 2: Formal context of *Concept_2*.

	Class CopyText Copy	**Class PasteText Paste**	**Method CopySettings CopyText**	**Method PasteSetting PasteText**	**Package Copy**	**Package Paste**
Class CopyText Copy	×		×		×	
Class PasteText Paste		×		×		×
Method CopySettings CopyText	×		×		×	
Method PasteSetting PasteText		×		×		×
Package Copy	×		×		×	
Package Paste		×		×		×

For the text editor example, the AOC-poset of Figure 4 shows two atomic blocks of variation (that correspond to two distinct features) mined from a single block of variation (*Concept_2* from Figure 3). The same feature mining process is used for the CB and for each of the BV.

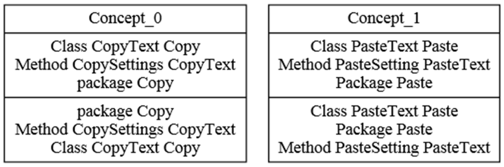

Figure 4: Atomic Blocks Mined from *Concept_2*.

5 Experimentation

To validate our approach, we ran experiments on two Java open-source softwares: Mobile Media[3] and ArgoUML[4]. We used 4 variants for Mobile Media, 10 for ArgoUML. The advantage of having two case studies is that they implement variability at different levels. In addition, Mobile Media and ArgoUML variants are well documented and their feature model is available for comparison to our results and validation of our proposal. Table 3 summarizes the obtained results for each software product variant. For readability's sake, we manually associated feature names to atomic blocks, based on the study of the content of each block and on our knowledge on software. Of course, this does not impact the quality of our results.

Table 3: Features Mined from Mobile Media and ArgoUML Softwares

Case Study	Feature		Evaluation Metrics			
Mobile Media Features	Common	Optional	K	Precision	Recall	F-Measure
Album Management	×		0.05	83%	62%	70%
Splash Screen	×		0.05	71%	57%	63%
Create Album	×		0.05	81%	58%	67%
Delete Album	×		0.05	80%	62%	69%
Create Photo	×		0.05	81%	52%	63%
Delete Photo	×		0.05	78%	63%	69%
View Photo	×		0.05	87%	68%	76%
Exception handling	×	×	0.03	100%	70%	82%
Edit Photo Label		×	0.02	100%	77%	87%
Favourites		×	0.04	100%	80%	88%
Sorting		×	0.06	100%	78%	87%
ArgoUML Features	Common	Optional	K	Precision	Recall	F-Measure
Class Diagram	×		0.03	72%	56%	63%
Diagram		×	0.06	100%	80%	88%
Deployment Diagram		×	0.05	100%	74%	85%
Collaboration Diagram		×	0.06	100%	67%	80%
Use Case Diagram		×	0.03	100%	64%	78%
State Diagram		×	0.03	100%	69%	81%
Sequence Diagram		×	0.02	100%	67%	80%
Activity Diagram		×	0.06	100%	63%	77%
Cognitive Support		×	0.01	100%	70%	82%
Logging		×	0.02	100%	60%	75%

Results show that precision appears to be high for all optional features. This means that all mined OBEs grouped as features are relevant. This result is due to search space reduction. In most cases, each BV corresponds to one and only one feature. For mandatory features, precision is also quite high thanks to our clustering technique that identifies ABVs based on FCA and LSI. However, precision is

[2] Here, intents and extents are the same. This is because the similarity matrix (and, consequently, the formal context) is symmetric.

[3] http://homepages.dcc.ufmg.br/~figueiredo/spl/

[4] http://argouml-spl.tigris.org/

smaller than the one obtained for optional features. This deterioration can be explained by the fact that we do not perform search space reduction for the CB. Considering the recall metric, its average value is 66% for Mobile Media and 67% for ArgoUML. This means most OBEs that compose features are mined. We have manually identified OBEs which should have been mined and were not. We found that these non-mined OBEs used different vocabularies from mined OBEs'. This is a known limitation of LSI which is based on lexical similarity. Considering the F-Measure metric, our approach has values that range from 63% to 88%. This means that most OBEs that compose features are mined and shows the efficiency of our approach. The most important parameter to LSI is the number of chosen term-topics (*i.e.*, Number of topics (*K*)). A term-topic is a collection of terms that co-occur frequently in the documents of the corpus. We need enough term-topics to capture real term relations. In our work we cannot use a fixed number of topics for LSI because we have blocks of variation (*i.e.*, partitions) with different sizes. The column (*K*) in Table 3 shows the K value for each feature.

6 Related work

In our previous work [8] we present an approach for feature location in a collection of software product variants based on FCA by distinguishing between the *common block* (*i.e.*, CB) and *blocks of variation* (*i.e.*, BVs). In this paper we extended our previous work by distinguishing between the common features that appear in the *common block* and the optional features that appear in the same *block of variation* based on the lexical similarity between OBEs. An inclusive survey about approaches linking features and sources code in a single software is proposed in [9]. The approach proposed by Ziadi *et al.* [4] is the closest to ours. They identify all common features as a single mandatory feature. Moreover, they do not distinguish between optional features that appear together in a set of variants. Their approach doesn't consider the method body. Rubin *et al.* [10] present an approach to locate optional features from two product variants' source code. They do not consider common features. They also are limited to only two variants. Xue *et al.* [3] propose an automatic approach to identify the traceability link between a given collection of features and a given collection of source code variants. They thus consider feature descriptions as an input.

7 Conclusion

In this paper, we proposed an approach based on FCA and LSI to mine features from the object-oriented source code of software product variants. We have implemented our approach and evaluated its produced results on two case studies. Results showed that most of the features were identified. The threat to the validity of our approach is that developers might not use the same vocabularies to name OBEs across software product variants. This means that lexical similarity may be not reliable in all cases to identify common and variable features. In future work, we plan to combine both textual and semantic similarity measures to be more precise in determining feature implementation.

References

[1] P. C. Clements and L. M. Northrop, *Software product lines: practices and patterns*. Addison-Wesley, 2001.

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

[3] Y. Xue, Z. Xing, and S. Jarzabek, "Feature location in a collection of product variants," in *19th WCRE Conference*. IEEE, 2012, pp. 145–154.

[4] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane, "Feature identification from the source code of product variants," in *15th CSMR Conference*. IEEE, 2012, pp. 417–422.

[5] B. Ganter and R. Wille, *Formal Concept Analysis, Mathematical Foundations*. Springer-Verlag, 1999.

[6] A. Marcus and J. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *25th ICSE Conference*. IEEE Computer Society, 2003, pp. 125–135.

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," November 1990.

[8] R. AL-Msie'deen, A. D. 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 '13 Conference*. Springer, 2013, pp. 302–307.

[9] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, pp. 1–54, 2012.

[10] J. Rubin and M. Chechik, "Locating distinguishing features using diff sets," in *27th ASE Conference*, ser. ASE 2012. ACM, 2012, pp. 242–245.