Mining Feature Models from the Object-Oriented Source Code of a Collection of Software Product Variants

Ra'Fat Ahmad Al-Msie'Deen

▶ To cite this version:

Ra'Fat Ahmad Al-Msie'Deen. Mining Feature Models from the Object-Oriented Source Code of a Collection of Software Product Variants. ECOOP: European Conference on Object-Oriented Programming, Jul 2013, Montpellier, France. pp.1-10. lirmm-01004928

HAL Id: lirmm-01004928

https://hal-lirmm.ccsd.cnrs.fr/lirmm-01004928

Submitted on 11 Jun 2014

Mining Feature Models from the Object-Oriented Source Code of a Collection of Software Product Variants

Ra'fat AL-msie'deen

LIRMM / CNRS & Montpellier 2 University, France
Al-msiedee@lirmm.fr

Abstract. In order to migrate software product variants which are considered similar into a software product line (SPL), it is essential to identify the mandatory and optional features between the product variants. To exploit existing software variants and build a 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 and feature models from the object-oriented source code of a set of software variants based on Formal Concept Analysis and Latent Semantic Indexing.

Keywords: Software product line engineering, software product variants, feature mining, feature model, Formal Concept Analysis, Latent Semantic Indexing.

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]. 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 further implies to identify the software family's common and variable features. Manual reverse engineering of the feature model for the existing software variants is time-consuming, error-prone, and requires substantial effort [2]. This paper proposes an approach to mine features from a collection of software product variants. This is the first step towards defining the feature model (FM) of the software family. Our approach is based on the identification of the implementation of these features among object-oriented (OO) elements of the source code (*cf.* Figure 1; where I_i are source code elements and F_j are features). Also in this paper, we mine feature model from software configurations (*i.e.*, product feature sets) that are produced by our previous works [1, 4].

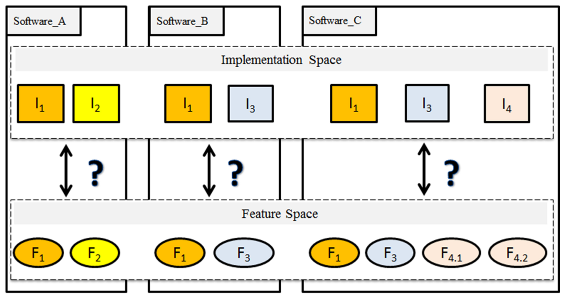

Fig. 1. Mining Feature From Software Product Variants

In our work two techniques are used for feature mining approach. The first one is FCA technique (*cf.* Section 3.1 and Section 3.2.2); FCA is a classification technique that takes data sets of objects and their attributes, and extracts relations between these objects according to the attributes they share [6]. The interested reader can find more information about our use of FCA in [1, 4]. The second technique is LSI (*cf.* Section 3.2.1); LSI is an advanced IR method. LSI computes textual similarity among different documents [5]. In LSI technique, one type of software artifact is treated as a query and another type of artifact is treated as a document. The textual similarity is computed based on the occurrences of terms in documents. If two documents share a large number of terms, those documents are considered to be similar. For more details about LSI the reader can refer to [4].

Our approach is detailed in the remainder of this paper as follows. Section 2 shows an overview of our approach. Section 3 presents the feature mining process. Section 4 describes the experiments that were conducted to validate our proposal. Section 5 discusses the related work. Section 6 concludes and provides perspectives for this work.

2 Approach Overview

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.

2.1 Features versus Object-oriented Building Elements

In this paper we focus on the mining of functional features [1, 4]. 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) and we restrict our study to *packages*, *classes*, *attributes*, *methods* or *method body elements* (*i.e.*, *local variable*, *attribute access*, *method invocation*). We consider that a feature corresponds to one and only one set of OBEs. We also consider that feature implementations may overlap: a given OBE can be shared between several features' implementation. Mining a feature from the source code of variants amounts to identify a 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). As the number of OBEs is large, mining features requires to reduce this search space (*cf.* Figure 2). Several strategies can be combined to do so:

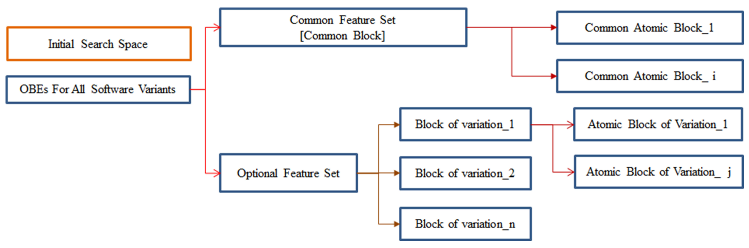

Fig. 2. The search space for feature mining

- separate the OBE set in two subsets, the common feature set – also called *common block* (CB) – and the optional feature set, on which the same search process will have to be performed.
- 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).
- 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 mapping model of Figure 3.

2.2 An Illustrative Example

As an illustrative example, we consider five database software variants. *Database system 1* supports core database feature: *notify triggers*. *Database system 2* supports *trigger list* and *state* features, together with the core feature. *Database system 3* supports *trigger list*, *state* and *acquire read lock* features, together with

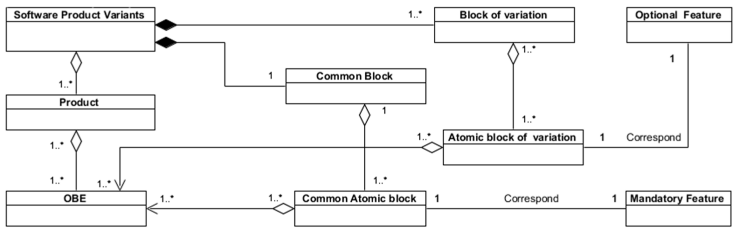

Fig. 3. OBE to Feature Mapping Model

the core feature. *Database system 4* has the core database feature and a new *release read lock* feature. *Database system 5* supports *trigger list*, *state*, *acquire read lock* and *release read lock* features, together with the core ones. In this example, we only use the source code of software variants as input of the feature mining process.

3 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 3.1). In the second step, we rely on LSI to determine the similarity between OBEs (*cf.* Section 3.2.1). This similarity measure is used to identify atomic blocks based on OBE clusters in the third step (*cf.* Section 3.2.2). Figure 4 shows our feature mining process.

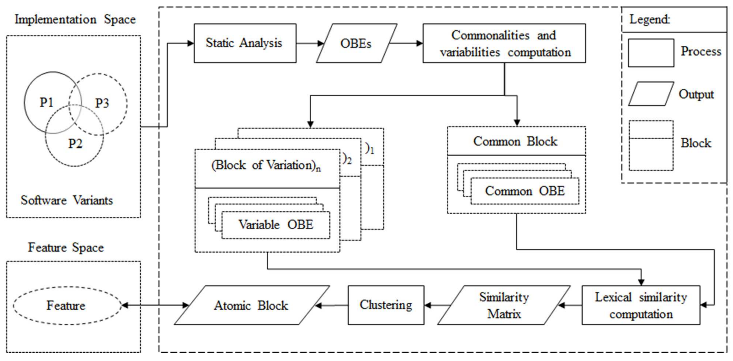

Fig. 4. The Feature Mining Process

3.1 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 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.* Figure 5 (left)), is defined. The corresponding AOC-poset is then calculated (*cf.* Figure 5 (right)). More information about this step can be found in [4].

	Package (com.database)	Class (Database_ com.database)	Method (notifyTriggers(locker)_ Database)	Attribute (triggerList_ Database)	Attribute Access (triggerList_ notifyTriggers(locker))	Local Variable (i_ notifyTriggers(locker))	Attribute (state_ Database)	Method (DatabaseState()_ Database)	Method (releaseReadLock()_ Database)	Method Invocation (releaseReadLock []_ notifyTriggers(locker))	Method (acquireReadLock()_ Database)	Method Invocation (acquireReadLock []_ notifyTriggers(locker))
System_1	×	×	×									
System_2	×	×	×	×	×	×	×	×				
System_3	×	×	×	×	×	×	×	×			×	×
System_4	×	×	×						×	×		
System_5	×	×	×	×	×	×	×	×	×	×	×	×

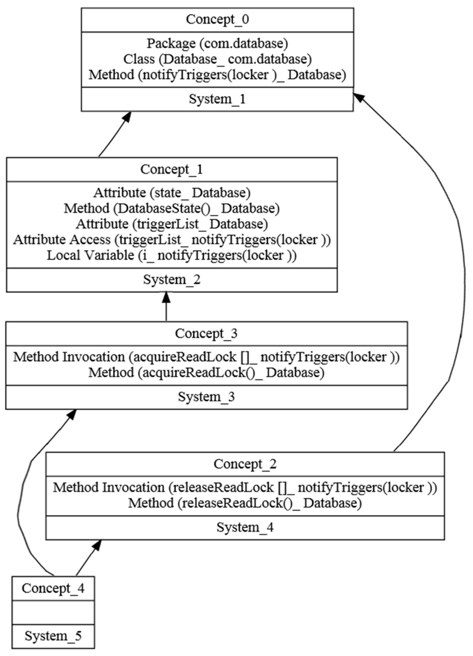
Concept_0
Package (com.database)
Class (Database_ com.database)
Method (notifyTriggers(locker)_ Database)
System_1
Concept_1
Attribute (state_ Database)
Method (DatabaseState()_ Database)
Attribute (triggerList_ Database)
Attribute Access (triggerList_ notifyTriggers(locker))
Local Variable (i_ notifyTriggers(locker))
System_2
Concept_3
Method Invocation (acquireReadLock []_ notifyTriggers(locker))
Method (acquireReadLock()_ Database)
System_3
Concept_2
Method Invocation (releaseReadLock []_ notifyTriggers(locker))
Method (releaseReadLock()_ Database)
System_4
Concept_4
System_5

Fig. 5. The Formal Context and AOC-poset for Database Software Variants

3.2 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.

3.2.1 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 hypothesis 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.* The interested reader can find more information about these three steps in [4].

#	1 Attribute (state_ Database)	2 Attribute (triggerList_ Database)	3 Attribute Access (triggerList_ notifyTriggers(locker))	4 Local Variable (i_ notifyTriggers(locker))	5 Method (DatabaseState()_ Database)
1	1	0.6173	0.0064	-0.1261	0.9997
2	0.6173	1	0.7906	0.7025	0.6350
3	0.0064	0.7906	1	0.9911	0.0292
4	-0.1261	0.7025	0.9911	1	-0.1035
5	0.9997	0.6350	0.0292	-0.1035	1

#	1 Attribute (state_ Database)	2 Attribute (triggerList_ Database)	3 Attribute Access (triggerList_ notifyTriggers(locker))	4 Local Variable (i_ notifyTriggers(locker))	5 Method (DatabaseState()_ Database)
1	x				x
2		x	x	x	
3		x	x	x	
4		x	x	x	
5	x				x

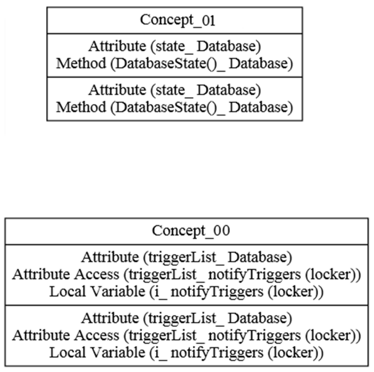
Concept_01
Attribute (state_ Database)
Method (DatabaseState()_ Database)
Attribute (state_ Database)
Method (DatabaseState()_ Database)
Concept_00
Attribute (triggerList_ Database)
Attribute Access (triggerList_ notifyTriggers (locker))
Local Variable (i_ notifyTriggers (locker))
Attribute (triggerList_ Database)
Attribute Access (triggerList_ notifyTriggers (locker))
Local Variable (i_ notifyTriggers (locker))

Fig. 6. Similarity matrix, Formal context and Atomic blocks mined from *Concept_1* of Figure 5 (right))

3.2.2 Identifying Atomic Blocks Using FCA: We then use FCA to identify, from each block of OBEs, which elements are mutually 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 [5]) meaning that only pairs of OBEs having a calculated similarity greater than or equal to 0.70 are considered similar. Figure 6 (left) shows the formal context obtained by transforming the similarity matrix corresponding to the BV of *Concept_1* from Figure 3 (right). As an example, in the formal context of this table, the OBE *"Attribute (state_ Database)"* is linked to the OBE *"Method (DatabaseState()_ Database)"* because their similarity equals 0.9997, which is greater than the threshold. However, the OBE *"Attribute (triggerList_ Database)"* and the OBE *"Method (DatabaseState()_ Database)"* are not linked because their similarity equals 0.6350, which is less than the threshold. The resulting AOC-poset is composed of concepts the extent and intent of which group similar OBEs. For the database example, the AOC-poset of Figure 6 shows two new concepts which will give rise to two atomic blocks of variation. They correspond to two distinct features mined from a single block of variation (*Concept_1* from Figure 5 (right)). The same feature mining process is used for the CB and for each of the BVs.

4 Experimentation

To validate our approach, we ran experiments on two Java open-source softwares: Mobile media[1] and ArgoUML[2]. Mobile media and ArgoUML variants are presented in Table 1 characterized by metrics LOC (Lines of Code), NOP (Number of Packages), NOC (Number of Classes) and NOOBE (Number Of Object-oriented Building Elements).

Table 1. Mobile Media and ArgoUML software product variants

Product Number	*Mobile Media Product Description*	*LOC*	*NOP*	*NOC*	*NOOBE*
P1	Mobile photo core	1,046	6	15	822
P2	Exception handling enabled	1,159	7	24	925
P3	Sorting and edit photo label enabled	1,314	7	25	1,040
P4	Favourites enabled	1,363	7	25	1,066
Product Number	*ArgoUML Product Description*	*LOC*	*NOP*	*NOC*	*NOOBE*
P1	All Features disabled	82,924	55	1,243	74,444
P2	All Features enabled	120,348	81	1,666	100,420
P3	Only Logging disabled	118,189	81	1,666	98,988
P4	Only Cognitive disabled	104,029	73	1,451	89,273
P5	Only Sequence diagram disabled	114,969	77	1,608	96,492
P6	Only Use case diagram disabled	117,636	78	1,625	98,468
P7	Only Deployment diagram disabled	117,201	79	1,633	98,323
P8	Only Collaboration diagram disabled	118,769	79	1,647	99,358
P9	Only State diagram disabled	116,431	81	1,631	97,760
P10	Only Activity diagram disabled	118,066	79	1,648	98,777

Table 2 summarizes the obtained results for each software product variant. For readability's sake, we manually associated feature names to atomic blocks. In order to evaluate our approach and based on our knowledge about software variants and their features (*i.e.*, OBEs for each feature) we have used three measures: precision, recall and F-Measure(*cf.* Equations 1, 2 and 3). All measures have values between [0, 1] [4].

$$Precision = \frac{\sum_i correctly\ retrieved\ OBEs}{\sum_i total\ retrieved\ OBEs}\% \qquad (1)$$

$$Recall = \frac{\sum_i correctly\ retrieved\ OBEs}{\sum_i total\ relevant\ OBEs}\% \qquad (2)$$

[1] http://homepages.dcc.ufmg.br/~figueiredo/spl/
[2] http://argouml-spl.tigris.org/

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

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. For mandatory features, precision is also quite high. 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. The most important parameter to LSI is the number of chosen term-topics (*i.e.*, Number of topics (K)). 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.

Table 2. Features Mined from Mobile Media and ArgoUML Softwares

Case Study	*Feature Type*		*# OBEs*		*Evaluation Measures*			
Mobile Media Features	*Common*	*Optional*	*Relevant OBEs*	*Correctly Retrieved OBEs*	*K*	*Precision*	*Recall*	*F-Measure*
Create Album	×		40	35	0.05	58%	87%	70%
Delete Album	×		35	30	0.05	60%	85%	70%
Create Photo	×		30	25	0.05	62%	83%	71%
Delete Photo	×		64	55	0.05	68%	85%	76%
View Photo	×		51	45	0.05	64%	88%	74%
Edit Photo Label	×		169	131	0.02	100%	77%	87%
Exception handling		×	104	73	0.03	100%	70%	82%
Favourites		×	58	45	0.06	100%	78%	87%
Sorting		×	32	26	0.04	100%	80%	88%
ArgoUML Features	*Common*	*Optional*	*Relevant OBEs*	*Correctly Retrieved OBEs*	*K*	*Precision*	*Recall*	*F-Measure*
Class Diagram	×		3587	2040	0.03	72%	56%	63%
Diagram		×	1309	1040	0.06	100%	80%	88%
Deployment Diagram		×	1334	1000	0.05	100%	74%	85%
Collaboration Diagram		×	935	630	0.06	100%	67%	80%
Use Case Diagram		×	1928	1250	0.03	100%	64%	78%
State Diagram		×	2597	1800	0.03	100%	69%	81%
Sequence Diagram		×	3708	2500	0.02	100%	67%	80%
Activity Diagram		×	1583	1000	0.06	100%	63%	77%
Cognitive Support		×	10193	7200	0.01	100%	70%	82%
Logging		×	1149	700	0.02	100%	60%	75%

Figure 7 part (a) shows the mined feature model (FM) for ArgoUML-SPL by our approach. The mined FM consists of optional and mandatory features with only one level of hierarchy and without cross-tree constraints and groups of features constraints. Our approach mines all concrete features from ArgoUML-SPL and Mobile Media software variants (*cf.* Figure 8). The FM in Figure 7 part (b) represents the FM as manually designed by the authors of ArgoUML-SPL. All abstract features in ArgoUML-SPL and Mobile Media software variants like feature "Album Management" or "Photo Management" are not mined by our approach. The abstract feature is not a concrete feature in the source code. It corresponds to a group of features or to the root feature in the FM. The abstract feature cannot be identified by our approach. In this paper, we mine FM from software configurations (*i.e.*, product feature sets) that were produced by our previous works [1, 4]. We integrated the FeatureIDE[3] plugin with our approach to represent the mined FM [4].

[3] `http://www.fosd.de/featureide/`

[4] `Available at https://code.google.com/p/refm/`

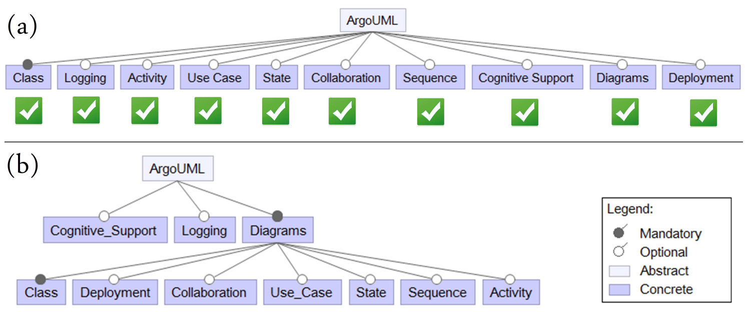

Fig. 7. The mined feature model for ArgoUML-SPL

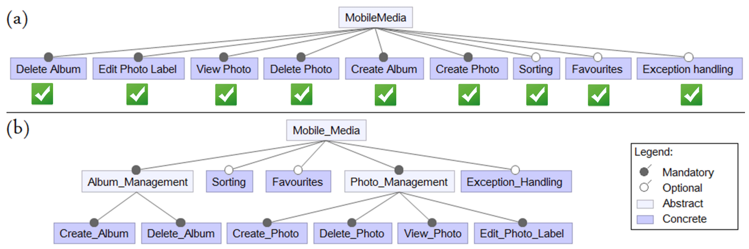

Fig. 8. The mined feature model for Mobile Media software variants

5 Related work

In our previous work [1] 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). We extended our previous work [1] by splitting blocks of source code elements based on the lexical similarity [4], in this second approach we distinguish 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. In this paper, we give more details about the mined features compared to the existing FM. An inclusive survey about approaches linking features and source code in a single software is proposed in [7]. Rubin *et al.* [8] present an approach to locate optional features from two product variants' source code. They do not consider common features and limit their proposal to two variants. Xue *et al.* [9] propose an automatic approach to identify the traceability links between a given collection of features and a given collection of source code variants. They thus consider feature descriptions as an input. The approach proposed by Ziadi *et al.* [2] 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.

6 Conclusion and perspectives

In this paper, we proposed an approach based on FCA and LSI to mine features and FM 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 mined FM represents all concrete features. 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 structural similarity measures to be more precise in determining feature implementation. In this paper, we manually associated feature names to atomic blocks, based on the study of the content of each block. As a future work we plan to automatically propose feature names for the atomic blocks. We also plan to mine *FM* directly from software configurations (*i.e.*, product feature sets) with its constraints (*i.e.*, cross-tree constraints and groups of features constraints).

References

1. AL-Msie'deen, R., Seriai, A.D., Huchard, M., Urtado, C., Vauttier, S., Salman, H.E.: Feature location in a collection of software product variants using formal concept analysis. In: ICSR '13 Conference, Springer (2013) 302–307
2. Ziadi, T., Frias, L., da Silva, M.A.A., Ziane, M.: Feature identification from the source code of product variants. In: CSMR'2012. (2012) 417–422
3. Arévalo, G., Berry, A., Huchard, M., Perrot, G., Sigayret, A.: Performances of galois sub-hierarchy-building algorithms. In: ICFCA. (2007) 166–180
4. AL-Msie'deen, R., Seriai, A., Huchard, M., Urtado, C., Vauttier, S., Salman, H.E.: Mining features from the object-oriented source code of a collection of software variants using fca and latent semantic indexing. In: SEKE '25 Conference. (2013)
5. Marcus, A., Maletic, J.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: ICSE '03 Conference. ICSE '03, Washington, DC, USA, IEEE Computer Society (2003) 125–135
6. Ganter, B., Wille, R.: FCA, Mathematical Foundations. Springer-Verlag (1999)
7. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process (2012) 53–95
8. Rubin, J., Chechik, M.: Locating distinguishing features using diff sets. In: 27th ASE Conference. ASE 2012, ACM (2012) 242–245
9. Xue, Y., Xing, Z., Jarzabek, S.: Feature location in a collection of product variants. In: 19th RE Conference, IEEE (2012) 145–154