Feature-Level Change Impact Analysis Using Formal Concept Analysis

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony

Feature-Level Change Impact Analysis Using Formal Concept Analysis

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony
UMR CNRS 5506, LIRMM, University of Montpellier 2 for Sciences and Technology, France
E-mail: {Eyalsalman, Seriai, Dony}@lirmm.fr

Abstract

Software Product Line Engineering (SPLE) is a systematic reuse approach to develop a short time-to-market and quality products, called Software Product Line (SPL). Usually, the SPL is not developed from scratch but it is developed by reusing features (resp. their source code elements) of existing similar systems developed by ad-hoc reuse techniques. The feature implementations may be changed for adapting SPLE context. The change may impact other features that are not interested in the change, as a feature's implementation spans multiple code elements and shares code elements with other features. Therefore, feature-level Change Impact Analysis (CIA) is needed to predict affected features for change management purpose. In this paper, we propose a feature-level CIA technique using formal concept analysis. In our experimental evaluation using three case studies of different domains and sizes, we show the effectiveness of our technique in terms of the most commonly used metrics on the subject.

Keywords: *Impact analysis, feature, source code, FCA, software product line, product variants.*

1 Introduction

SPLE is a engineering discipline providing methods to promote systematic software reuse for developing short time-to-market and quality products in a cost-efficient way [1]. These products are referred to as SPL [1]. The whole idea behind SPLE is to build core assets consisting of all reusable software artifacts (such as source code, test cases and son) that can be leveraged to develop SPL's products. Building such core assets is driven by features that SPL products should provide. A feature is "a prominent or distinctive user-visible aspect, quality or characteristic of a software system" [2].

Building SPL's core assets from scratch is a costly task [1]. Therefore, features (resp. their source code elements) of existing similar systems developed by ad-hoc reuse techniques should be reused as much as possible to build SPL's core assets [3]. Such systems are known as *product variants*. The implementation of the obtained feature(s) may need to be changed for adapting SPLE context by adding or removing requirements (resp. their source code elements) [4]. The change may impact other features that are not interested in the change, as a feature's implementation spans multiple code elements (e.g., classes and methods) and shares code elements with other features. To avoid such a situation, feature-level Change Impact Analysis (CIA) is needed to detect introducing undesirable interactions between feature implementations. Furthermore, it is helpful to conduct change management from a SPL manager's point of view. For example, managers may most likely be interested in evaluating a given source code change in terms of affected features, in order to decide which change strategy should be executed.

Feature-level CIA is far from a trivial task when we have a large number of features. Manually tracing feature implementations to determine affected features is time-consuming, error-prone and tedious. The CIA is seldom considered at the feature level for changes made to the source code level. Most of the existing works performs CIA at the source code level with little works completed at requirement and design levels [5]. In this paper, we propose a technique to study CIA at the feature level using Formal Concept Analysis (FCA). This technique takes, as input, a change set composed of classes to be changed and computes, as output, a ranked list of affected features. Each feature in this list has a probability to be affected representing the feature priority to be checked by maintainers. Additionally, we propose two metrics to measure to what degree a given feature implementation is impacted and the changeability of the features.

The rest of this paper is organized as follows. Section 2 presents FCA. Sections 3 and 4 present the proposed approach and experimental evaluation, respectively. Next, sections 5 and 6 discuss the related works and conclude the paper respectively.

2 Background: Formal Concept Analysis

FCA is a technique for data analysis and knowledge representation based on lattice theory [6]. Concept lattices are core structures of FCA for extracting a set of concepts from

Table 1. A formal context for birds.

	Flying	Nocturnal	Feathered	Migratory	with-crest	with-membrane
Flying-squirrel	X					X
Bat	X	X				X
Ostrich			X			
Flamingo	X		X	X		
Chicken	X		X		X	

a dataset, called a formal context, composed of objects described by attributes. A formal context is defined as a triple $K = (O, A, R)$ where O is a set of objects, A is a set of attributes and R is a binary relation between objects and attributes indicating which attributes are possessed by each object. Table 1 presents an example of a formal context for several animals described by their characteristics.

A formal concept is a pair (E, I) composed of an object set ($E \subseteq O$) and an attribute set ($I \subseteq A$). E is the extent of the concept (i.e., the objects covered by the concept). I is the intent of the concept (i.e., the attributes shared by the objects covered by the concept). For example, ($\{$*Chicken, Flamingo*$\}\{$*Flying, Feathered*$\}$) is a concept of our example.

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 $C_1 = (E_1, I_1) \leq_s C_2 = (E_2, I_2)$ 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. Based on this specialization order definition, an important property is that a sub-concept inherits in a top-down manner the attributes (intent) of its super-concepts, while a super-concept inherits in a bottom-up manner the objects (extent) of its sub-concepts.

Let C_K be the set of all concepts of a formal context K. This set of concepts provided with the specialization order (C_K ,$\leq_s$) has a lattice structure, and is called the concept lattice associated with K. Figure 1 shows the concept lattice built for the formal context of Table 1.

3 The Proposed Approach

When source code classes that implement a feature are changed, the change may be propagated to the neighbor classes which the changed classes are coupled to. As a feature's implementation may span multiple classes and features may have shared classes, the change may lead to impact the implementation of other features. Therefore, we should determine the impact set of classes and coupling relations between features to determine affected features. We rely on structural and feature couplings to support these purposes respectively. Structural coupling refers to interdependencies between classes, such as inheritance, method invocation, etc. Feature coupling is the degree to which the source code elements implementing a feature (e.g., methods, attributes, classes) depend on elements outside the feature [7]. Figure 2 gives an overview of our approach. This approach relies on three main steps:(i) computing the impact set of classes for source code changes, (ii) Determining coupled features using FCA, (iii) querying the generated concept lattice to compute a ranked list of affected features. The goal of the first step is to determine impact set classes due to modifying a given change set of classes. Of course, the impact set also includes the change set members. The second step aims to determine coupled features by building the concept lattice. This lattice is queried in the third step using the impact set computed in the first step to find a ranked list of affected features.

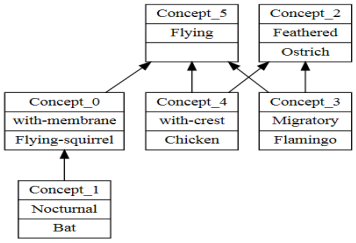

Figure 1. The concept lattice for the formal context of Table 1.

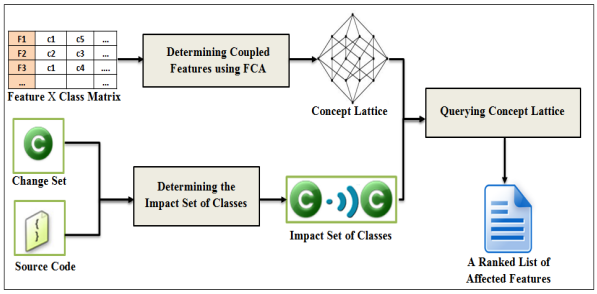

Figure 2. Main steps of our CIA approach.

3.1 Determining the Impact Set of Classes

Analyzing the interdependencies between classes helps to determine the coupled classes, and hence determine the impact set of classes. We rely on the following interdependencies that represent coupling aspects in object-oriented applications supporting CIA: (1) *inheritance relationship*: when a class inherits attributes and methods of another class, (2) *method call*: when a method of one class calls a method of another class, (3) *attribute access*: when a class accesses an attribute of another class, (4) *shared attribute access*: when two classes access the same attribute of another class. To capture these interdependencies, the given source code is statically analyzed by building an abstract syntax tree (AST) that can be queried to extract required information.

Table 2. The formal context of features and classes.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
F1	X	X	X	X	X							
F2	X	X	X			X			X			
F3	X	X			X		X					
F4	X		X			X						
F5	X				X		X	X	X			
F6										X	X	
F7										X		X

3.2 Determining Coupled Features Using FCA

In this step, we rely on FCA to analyze coupling relations between a given set of features. FCA allows us to determine and visualize source code classes that are shared among all features, among a subset of features and those that are specific to each feature through the hierarchical organization of the concept lattice. This step takes as input a feature-to-class traceability matrix. This matrix can be obtained by our previous work [8]. In this matrix, each feature is linked to its implementing classes. Columns and rows respectively represent features and source code classes. This matrix represents the formal context where features and classes respectively represent objects and attributes. The relation between an object and an attribute refers to which feature is implemented by which class. According to this definition of the formal context, we can obtain a concept lattice containing concepts that are composed of a set of features sharing a set of classes. Such a lattice represents dependencies between features and classes (feature coupling). Table 2 is an example of the formal context to be analyzed where objects are $\{F1, F2, F3, F4, F5, F6, F7\}$ while attributes are $\{C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12\}$. Figure 3 shows the corresponding lattice of the context shown in Table 2. Based on this lattice we discover the following observations:

- The impact of changes made to classes of concepts that are located at the top of the lattice is propagated to all extents (features) of these concepts. This is because the classes of such concepts are shared among all or most lattice concepts. Such classes should not be impacted as far as possible because they may lead to the risk of changing all features. For example, if *C1* and *C10* respectively at *Concept_7* and *Concept_13* are modified or impacted, all features (F1 to F7) will be affected.

- The impact of changes made to classes of lattice concepts located at the bottom is local. For example, if *C8* is modified or impacted, only $F5$ will be affected. Determining these classes is useful to guide the maintainers to choose from available change strategies, the one that considers only such classes to implement the change request.

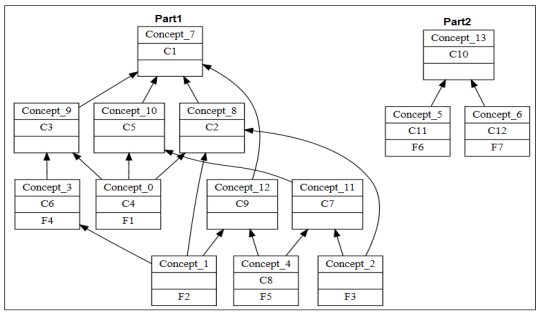

Figure 3. The concept lattice for the formal context of Table 2.

- By descending vertically throughout the lattice, the impact of changes is gradually decreased. For example, if changes are made to $\{C1\}$, the set of affected features will be composed of $\{F1, F2, F3, F4, F5\}$, but if changes are made to $\{C7\}$, the set of affected features will only be composed of $\{F3, F5\}$.

- Lattice concepts that are downwardly reachable from jointly changed classes have a high probability of being affected. For example, assuming that impact set = $\{C2, C3, C5\}$, then $F1$ has a higher probability of being affected than $F2$. This is because $F1$ will be affected by three joint classes $\{C2, C3, C5\}$, while $F2$ will be affected by two joint classes $\{C2, C3\}$.

- Based on the generated lattice, we can determine isolated sub-systems/parts and hence determine maintenance resources that are required based on the affected subsystems/parts. For example, if the affected features are *F6* and *F7* (i.e., part2) and they are maintained by specific team, a product manager can exclude other maintenance teams and ask only the interested team to execute the change.

3.3 Querying Concept Lattice

3.3.1 Determining Affected Features

In this step, we query the generated lattice by the impact set of classes to retrieve features that are implemented by these classes. The retrieved features represent two subsets of affected features, following the two steps below. Firstly, locating a set of lattice concepts that have as intent (excluding inherited intent) one or more of impacted classes. The extents of these concepts are a subset of affected features. Secondly, determining all downwardly reachable concepts from concepts obtained in the step 1. The extents of these concepts are another subset of affected features.

The first step is performed using a simple algorithm to explore the concept lattice for determining the required concepts, called (*CON*). Due to space limitation, we are unable to present this algorithm. To perform the second step, we

propose algorithm 1. This algorithm is based on depth-first search (DFS). The algorithm takes as input a list of concepts computed in step 1 (*CON*) and the concept lattice to be queried (*CL*). Lines 1-10 check each concept in *CON* in turn so that each one becomes the root of a new tree in the DFS. The extents of concepts that constitute such a tree represent all affected features due to changes made to classes of its parent concept. The function *getAdjacentConcepts()* returns all concepts that are located immediately below and directly related to the current concept (*Co*). LC_F is an accumulator for all traversed concepts. These traversed concepts represent all downwardly reachable concepts from *CON*'s concepts. LC_F may contain some concepts that do not have their own extent and LC_F also can include redundant concepts due to the overlap between DFS's trees. Therefore, lines 11-14 remove these concepts respectively using the functions (*RemovingConHavingEmptyExt()*) and (*RemovingRedundantConcepts()*). In line 15, we extract the extent of LC_F concepts using the function (*ExtractingExtent()*) because the extent of these concepts represent the affected features.

Algorithm 1: LocatingAffectedFeatures

Input: $\mathcal{CON}$, $\mathcal{LC}$
Output: AF // list of affected features

1 $\mathcal{LC}_F \leftarrow \phi$
2 **foreach** *j from 1 to* $|\mathcal{CON}|$ **do**
3 $ConceptStack \leftarrow \mathcal{CON}[j]$
4 **while** $(ConceptStack$ *is not empty)* **do**
5 Co $\leftarrow$ pop($ConceptStack$)
6 AdjCos $\leftarrow$ getAdjacentConcepts(Co, CL)
7 **if** *(AdjCos is empty)* **then**
8 $\mathcal{LC}_F \leftarrow$ Co
9 **else**
10 push(AdjCos), $\mathcal{LC}_F \leftarrow$ Co
11 **foreach** i *from 1 to* $|\mathcal{LC}_F|$ **do**
12 **if** *Extent*($\mathcal{LC}_F[i]$) *is empty* **then**
13 RemovingConHavingEmptyExt($\mathcal{LC}_F$, i)
14 RemovingRedundantConcepts ($\mathcal{LC}_F$)
15 AF $\leftarrow$ ExtractingExtent($\mathcal{LC}_F$)
16 **return** *AF*

3.3.2 Ranking Affected Features

Based on the observations of concept lattice mentioned earlier, we can deduce that the concept lattice organizes features hierarchically according to their probability to be impacted by a given change proposal. Therefore, we propose two metrics adapted to feature-level CIA: *Impact Probability Metric* (IDM) and *Changeability Assessment Metric* (CAM).

IDM is used to measure the degree to which a specific feature can be affected. Features having high *IDM* values, the functional requirements provided by these features have a high probability to be affected. Therefore, their implementation should first be checked by maintainers. IDM values are in the range [0, 1]. Using IDM metric, we can rank the affected features from the feature that has a higher IDM value to the feature that has a lower IDM value. We propose the following equation for IDM:

$$IDM(F) = \frac{|\{I\} \bigcap \{\textit{Impact Set}\}|}{|\{I\}|} \times 100\% \qquad (1)$$

In Equation 1, F is an affected feature while I is the intent (classes) of a lattice concept having F as an extent and also includes intents inherited in a top-down manner. Thus, we need to compute the inherited intents of lattice concepts (LC_F) having the affected features. This is performed using the DFS algorithm to compute all upwardly reachable concepts from each concept in $\mathcal{LC}_F$.

CAM is a metric for determining the percentage of features that are affected by a given change. It describes the changeability of all features in order to help product managers to decide whether a change proposal is accepted or to find another change plan more suitable to employ. We propose the following equation for CAM:

$$\textit{CAM} = \frac{\#\textit{Affected\_Features}}{\#\textit{All\_Features}} \times 100\% \qquad (2)$$

In equation 2, *Affected_Features* represent a set of features that are potentially affected by a given impact set of classes. $All\_Features$ represent all features. CAM values take a range [0, 1]. If the computed CAM value is high, this means that features (resp. their implementation) are more sensitive for a given change proposal and vice versa.

By referring again to Figure 3 and considering that the impact set is composed of $\{C3, C5, C6\}$, we find out that the IDM and CAM values of this impact set as shown in Table 3. The columns (*Concept_No*, *Features* and *Rank*) show respectively a set of concepts having affected features, their affected features and the priority of these features to be checked. These features are ranked based on their IDM values. From Table 3, we notice that *F4* has the highest IDM value. Also, It shows that CAM for this impact set is equal to (71%). This means that most of features will be affected by this given impact set of classes.

4 Experimental Results and Analysis

4.1 Case Studies

For evaluation, we have applied our CIA technique respectively to core assets of three different case studies:

Table 3. Impact results for {C3, C5, C6} changes.

Concept_No	Features	$\|\{I\}\|$	$\|\{I\} \bigcap \{FCS\}\|$	IDM	Rank	CAM
Concept_3	F4	3	2	66%	1	71%
Concept_1	F2	5	2	40%	2	
Concept_0	F1	5	2	40%	2	
Concept_2	F3	4	1	25%	3	
Concept_4	F5	5	1	20%	4	

ArgoUML-SPL[1], *MobileMedia*[2] and *BerkelyDB-SPL*[3]. *MobileMedia* is a small-scale system for managing multimedia files on mobile devices. The core assets of *MobileMedia* supports *6* features such as, *view photo, delete photo, sort photos,* etc. *ArgoUML-SPL* is a large-scale system for UML modeling tool. The core assets of *ArgoUML-SPL* supports *8* features such as, the *Class* diagram, the *State* diagram, the *Activity* diagram, etc. *BerkeleyDB-SPL* is a large-scale embedded database system that can be embedded in other applications as a storage engine. The core assets of *BerkeleyDB-SPL* provides *25* features, such as *transaction management*, *concurrency control*, etc.

4.2 Evaluation Measures

We relied on three measures inspired from information retrieval, namely *precision*, *recall* and *F-measure* to evaluate our CIA technique. *Precision* measures the accuracy of estimated impact set of features (EIS) according to the actual impact set (AIS). The EIS represents the affected feature computed by our technique while AIS is computed by manually tracing features to identify the affected features for each given change proposal. *Recall* measures the degree to which the EIS covers the AIS members. F-measure makes a trade-off between precision and recall, so that it gives a high value only in the case that both recall and precision values are high. Based on the definitions above, we can deduce that precision also quantifies elements of EIS that actually are not impacted (false-positive). Also, recall quantifies the features that are not identified but are impacted (false-negatives). Our proposed technique aims to achieve high precision, recall and F-measure. All measures have values within [0,1]. If the EIS has a high precision, this means that maintainers spend less time and effort to locate affected features. If the EIS has a high recall, this gives maintainers the confidence that all of affected features have been considered.

4.3 Effectiveness of Our CIA Technique

Table 4 summarizes the results obtained by our CIA technique. Columns describe respectively: change set of classes (*CSC*), the size of CSC ($|CSC|$), the size of estimated impact set of features ($|EIS|$), *precision*, *recall*, *F-measure* and *CAM*. We randomly select three different *CSCs* for both *ArgoUML-SPL* and *MobileMedia*, and two *CSCs* for *BerkeleyDB-SPL*. Therefore, we have *8 CSCs* to be analyzed. These selected CSCs are modified by considering different change types, including changes made to *class signature*, *class body*, *attributes*, *method signature* and *method body*.

Table 4. Precision, Recall and F-measure of our CIA Technique.

CSC	$\|CSC\|$	$\|EIS\|$	Precision	Recall	F-measure	CAM
MobileMedia						
CSC1	5	5	60%	75%	67%	100%
CSC2	5	6	83%	100%	90%	83%
CSC3	8	6	67%	100%	80%	100%
ArgoUML-SPL						
CSC1	9	5	80%	100%	88%	62%
CSC2	8	4	75%	100%	86%	50%
CSC3	18	5	80%	100%	88%	62%
BerkeleyDB-SPL						
CSC1	6	25	92%	100%	96%	92%
CSC2	5	25	100%	100%	100%	100%

Table 4, shows that precision values are fluctuated and take a value in the range between 60% and 100%. This fluctuation can be attributed to two reasons. Firstly, some changes made to CSC do not have any impact on feature implementations. These changes include deleting dead source code (e.g., conditional branch that logically will never be entered), adding output statements, etc. Secondly, the impact set of classes may contain some classes that, in fact, are not impacted. Such classes are called *false-positive classes*. For example, consider that `C1` and `C2` are two classes connected by a method invocation and `C1` is proposed to be changed by adding an attribute. In this case, `C2` is considered as affected class in spite of it is not be affected by this change. Such a case indicates features that are implemented by false-positive classes actually are not affected. Recall values shown in Table 4 are high where these values take a range between 75% and 100%, and in most cases they reach 100%. The reason that hinder our technique to achieve 100% for all *CSCs* is that we do not consider classes that are not neighbors of CSCs. These classes may contribute to implement feature(s). F-measure values confirm that our CIA technique gives high precision and recall, where these values are high taking a range between 67% and 100%.

CAM values in Table 4 shows the changeability of features of each case study against each CSC considered. We notice that all CSCs of *MobileMedia* affect more than half of its features. This is because *MobileMedia* is a small-scale system, and hence its source code classes are strongly coupled so that any change may impact many different features. For *ArgoUML-SPL*, all changes made to its features affect

[1]http://argouml.tigris.org/
[2]http://www.ic.unicamp.br/ tizzei/mobilemedia/
[3]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/

almost half of its features. This is due to *ArgoUML-SPL*'s features being loosely coupled. They appear as isolated subsystems, for example, *cognitive* feature is implemented by 221 classes. For *BerkeleyDB-SPL*, changes made to its features affect most of its features. By investigating the impact set of classes and the concept lattice corresponding to this case study, we find that one of the changed classes (*EnvironmentImpl*) is located at the top of the lattice. This means that changes made to this class are propagated to the implementation of most features, which leads to a rise in the value of CAM. Based on CAM values shown in Table 4, we notice that these values quantify the changeability of the features against each change proposal. This allows SPL's manager to select the change strategy that results in the lowest possible CAM value.

5 Related Work

A large body of research is proposed for CIA. A comprehensive survey about CIA techniques can be found in [9]. The approach proposed by Revelle et al. [10] is the closet to ours. They proposed a feature coupling metric for supporting CIA at the feature level since features that are strongly coupled to a feature with modified implementation are the most likely to be affected. In their approach, features with a coupling value equal to or above a given threshold are considered as coupled features. However, coupled features under the specified threshold value may be affected by the change as shown in their experimental results. Hammad et al. [11] proposed an approach to determine which source code changes impact the system architecture. Diaz et al. [12] proposed an approach to perform CIA at the architectural level for changes induced at the requirement level. Chechik et al. [13] proposed a model-based approach for studying changes propagated between requirements and design models. Khan and Lock [14], utilized dependencies between requirement-level concerns and architectural components to studying the impact of requirement changes at the architecture level. All works mentioned above, except Revelle et al.'s work [10], do not support CIA at the feature level. They perform CIA at other different levels of abstraction.

6 Conclusion

In this paper, we have proposed a feature-level CIA approach to study the impact of changes made to source code of features obtained from product variants. This approach is useful for conducting change management from SPL manger's point of view. Our approach takes, as input, a change proposal at class level and computes a ranked list of potential affected features. Also, we propose two metrics to support this point of view. The proposed approach employed formal concept analysis, feature and structural couplings. Our experiments on three core assets of three case studies of different domains and sizes proved the effectiveness of our approach in terms of the most used metrics on the subject (*precision*, *recall* and F-*measure*).

References

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

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

[3] D. Beuche, "Transforming legacy systems into software product lines." in *SPLC*. IEEE, 2011, p. 361.

[4] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 112–121.

[5] L. Bixin, S. Xiaobing, L. Hareton, and Z. Sai, "A survey of code-based change impact analysis techniques," in *software testing, verification and reliability*. Wiley Online Library, 2012.

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

[7] S. Apel and D. Beyer, "Feature cohesion in software product lines: an exploratory study," ser. ICSE '11. ACM, 2011, pp. 421–430.

[8] H. Eyal-Salman, A.-D. Seriai, and C. Dony, "Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval," ser. IRI'13. California, USA: IEEE, 2013, pp. 209–216.

[9] D. L. Andrea, F. Fausto, and O. Rocco, "Traceability management for impact analysis." in *ICSM*, 2008, pp. 21–30.

[10] M. Revelle, M. Gethers, and D. Poshyvanyk, "Using structural and textual information to capture feature coupling in object-oriented software," *Empirical Softw. Engg.*, vol. 16, no. 6, pp. 773–811, 2011.

[11] M. Hammad, M. L. Collard, and J. I. Maletic, "Automatically identifying changes that impact code-to-design traceability during evolution," *Software Quality Control*, vol. 19, no. 1, pp. 35–64, 2011.

[12] J. Daz, J. Prez, J. Garbajosa, and A. L. Wolf, "Change impact analysis in product-line architectures." in *ECSA*, 2011, pp. 114–129.

[13] M. Chechik, W. Lai, S. Nejati, J. Cabot, Z. Diskin, S. Easterbrook, M. Sabetzadeh, and R. Salay, "Relationship-based change propagation: A case study," ser. MISE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 7–12.

[14] S. S. Khan. Simon Lock, "Concern tracing and change impact analysis: An exploratory study," ser. EA '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–48.