Identifying Traceability Links between Product Variants and Their Features

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony, Ra'Fat Ahmad Al-Msie'Deen

Identifying Traceability Links between Product Variants and Their Features

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony and Ra'fat Al-msie'deen
UMR CNRS 5506, LIRMM
Université Montpellier 2 Sciences et Techniques
Place Eugène Bataillon, Montpellier, France
{Eyalsalman, Seriai, Dony, Al-msiedeen}@lirmm.fr

***Abstract*— usually a software product line (SPL) is developed by exploiting available resources of a set of software variants that deem similar. In order to reengineer such variants that are developed by ad-hoc reuse into software product line that are developed by systematic reuse, it is necessary to identify traceability links between features and source code in a collection of product variants. Information retrieval (IR) methods are used widely to achieve this goal. These methods handle product variants as singular entities. However when product variants are considered together, we can get additional information that improves IR results. This paper proposes an approach to improve IR results when they are applied to identify traceability links in a collection of product variants. The novelty of our approach is that we exploit commonality and variability across product variants at feature and implementation levels to apply IR methods in efficient way. The obtained results proved that our approach significantly outperforms direct applying IR technique in conventional way in term of precision and recall metrics.**

Keywords- Traceability links, features, source code, object oriented, variability, software product line, latent semantic indexing, product variants.

I. INTRODUCTION

SPL aims to reduce development cost and time by producing a family of software products at a time. According to software engineering institute (SEI) definition, a SPL is "a set of software-intensive systems sharing a common, managed set of features that satisfies the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [3]. Usually SPL is developed by exploiting available resources of a set of software variants that deem similar to build SPL core assets. Such resource includes: source code, design documents, features and so on [8].

Software product variants represent a set of similar products that are developed by ad-hoc reuse techniques such as "clone-and-own". These variants share some features and also differ in others to meet specific needs of customers in a particular domain. For example, Wingsoft Financial Management System (WFMS) was developed for Fudan University and then evolved many times so that all evolved WFMS systems have been used in over 100 universities in China [5].

At first glance, clone-and-own technique represents an easy and fast reuse mechanism so that it provides the ability to start from existing already tested code and then making required modifications to produce a new variant. However when number of product variants and features grows, such ad-hoc reuse technique causes critical problems such as: maintaining efforts will be increased because we should maintain each variant separately from others, and sharing features in new products will be more complicated. When these problems accumulate, it is necessary reengineering product variants into a SPL for systematic reuse.

The first step in this reengineering process is to identify source code elements that implement a particular feature across product variants. This mapping between features and corresponding source code elements is known as traceability links.

The identified traceability links can be used to facilitate products derivation process from SPL core assets, find dependency between features, facilitate program comprehension process and also no maintenance task can be completed without identifying source code elements that are relevant to the task at hand[11].

Numerous approaches that are based on IR techniques have been proposed to identify links between source code and features [6]. These approaches handle product variants as singular entities (one product at a time). However when product variants are considered together, we can get additional information that can help to improve IR techniques results. This information is about commonality and variability across product variants at feature and implementation levels.

In this paper, we propose new approach to identify traceability links between object oriented source code of a set of product variants and given features of these variants using latent semantic indexing (LSI). Our approach aims to divide LSI search space at feature and implementation levels for each variant into two partitions (or subspaces): common and variable partitions. At features level, common partition represents a set of features that are shared by all variants (common features) while other features in the same variant represent variable partition (optional features). At implementation level, common partition refers to source code elements that realize common features while other implementation in the same variant represents variable partition that realize optional features. Source code elements implementing common features are called common source code elements while source code elements implementing optional features are called variable source code elements. The intuition behind this dividing process is to isolate common features and their corresponding code in each product variant. Consequently, we can also isolate optional features and their corresponding code in each product variant. The experimental results show that our approach

gives promised results comparing with applying LSI in conventional way (a variant as atomic chunk).

The remainder of this paper is organized as follows. Section 2 presents background and related work. Section 3 presents our approach. Section 4 shows experimental results and evaluation. Finally, Section 5 presents conclusion and future work.

II. BACKGROUND AND RELATED WORKS

This section describes LSI and traceability links in software engineering, and discusses related works.

A. Traceability Links

Traceability is the ability to describe and follow the life cycle of an artifact (requirements, design models, source code, etc.) created during the software life cycle in both forward and backward directions [13]. Traceability relations can refer to overlap, satisfiability, dependency, evolution, generalization/refinement, conflict or rationalization associations between various software artifacts. In general, traceability relations can be classified as horizontal or vertical relations. The former type refers to relation among artifacts at different levels of abstraction (e.g. between requirements and design) and the latter type refers to relation among artifacts at the same level of abstraction (e.g. among related requirements) [14].

Identifying traceability links among software artifacts at different levels of abstraction of product variants provide important information about development and maintenance a SPL. Such traceability is useful to derive concrete products from SPL core assets.

B. LSI in Software Engineering

Several IR methods exist such as: probabilistic method (PM), vector space method (VSM) and LSI [15]. All of these methods assume that all software artifacts are in textual format. In each method, one type of software artifact is treated as query and another type of artifact is treated as document. IR methods rank these documents against queries by extracting information about the occurrences of terms within them. The extracted information is used to find similarity between queries and documents. In the case of recovering traceability links, this similarity is exploited to recover traceability links that might exist between two artifacts, one of them is used as query.

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, etc.) and to overcome two classic problems arising in natural languages processing: synonymy and polysemy. The intuition behind SVD is rather complex to be presented here and see [16] for further details.

We chose LSI because it already has positive results to address maintenance tasks such as concept location [15], detection in software [17], and recovery of traceability links between source code and documentation [1].

C. Related Works

A comprehensive survey about techniques that have been proposed to identify source code elements relevant to a feature can be found in [9]. These techniques depend on static, dynamic or textual analysis, or a combination of these. Static analysis examines structural information such as control or data flow. Dynamic analysis relies on execution trace according to scenarios related to specific feature(s). Finally, textual techniques examine words in source code using IR methods. All these techniques identify traceability links between source code and corresponding features in a single product while our approach considers a set of product variants. The following works represent the most relevant works to us.

Ghanam et al. [4] have proposed a method to keep traceability links between feature model (FM) of a SPL and source code up-to-date. When SPL evolves, the traceability links become broken or outdated due to evolution at features and implementation levels. Their method is based on executable acceptance tests (EAT). EAT refers to English-like specifications (such as: scenario and story tests).These EATs represent the specifications of a given feature and can be executed against the system to test the correctness of its behavior. Their approach starts from already existing links to make them up to date while our approach is differ from this work where we start from scratch and assume that no already existing links.

Rubin et al. [12] focused on locating distinguishing features of two product variants realized via code cloning. Distinguishing features mean those features that are present in one variant and absent in another. Their approach relies on capturing the information about unshared part of the code between two products. This unshared part can be obtained by comparing a variant's source code that has the features of interest to another one that does not has. The distinguishing features between to variants reside on this unshared part of code. Their work aims at isolating source code that corresponds to distinguishing features and then apply feature location techniques in efficient way. However, if the number of distinguishing features is large their approach becomes infeasible because in this case we map a large number of features with a large part of code.

III. THE PROPOSED APPROACH

In this section, we describe input data of our approach; discuss how to divide each variant at feature and implementation levels into two partitions and how to apply LSI for recovering traceability links.

A. An Illustrative Example

Consider a collection of four variants of text editor system as shown in table 1 below. The initial product in this collection is T_Editor_V1.0. It supports just core features for any text editor such as: open, save and create a file. The initial product is enhanced to be *T_Editor_V1.1* by adding *search* and text *edit* features. *T_Editor_V1.2* is another enhancement of initial product. *T_Editor_V2.0* is an

advanced variant of text editor. It supports all previous features in addition to *replace* feature.

Table1. Features Set of Four Text Editor Variants.

Product Name	Features
T_Editor_V1.0	Core (Open, Save, Create).
T_Editor_V1.1	Core, Search, Edit.
T_Editor_V1.2	Core, Print, Edit.
T_Editor_V2.0	Core, Search, Edit, Replace, Print.

B. *Input Data*

Our approach takes object oriented source code of a set of product variants and a given set of features of these variants as input (like table 1). Each feature is identified by its name and description. Feature description is a natural language description. This information about feature represents a domain knowledge that is usually available from product variants documentation. In our work, feature description consists of small paragraph or some sentences.

C. *Feature versus Object-Oriented Elements*

In the literature, there are many definitions of feature. In this work, we rely on the following definition: a feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [19]. We adhere to the classification given by [18] which distinguishes three categories of features: firstly, functional features express the behavior or the way users may interact with a product. Secondly, interface features express the product's conformance to a standard or a subsystem. Finally, parameter features express enumerable, listable environmental or non-functional properties. In our work, we deal with functional aspects of features where functionalities are grouped together into at a high level of abstraction to form features.

As there are several ways to implement features [7], we assume that functional features either common or optional are implemented at the programming language level. Thus in an object oriented source code, functional feature can be implemented by different object oriented building elements (OOBEs). OOBEs include packages, classes, methods, attributes, etc. A feature has coarse granularity elements when its implementation consists of high level building units such as: packages, classes and interfaces. On the other hand, a feature is fine-grained when its code is composed by lower level units, such as methods, attributes, statements. In our work, we consider that a feature is realized at implementation level by a set of classes because the class represents a main building unit in any object oriented language.

D. *Identifying Common and Variable Partitions at Feature and Implementation Levels For Each Variant*

The goal of our approach is to reduce LSI search space as much as possible at feature and implementation levels. The

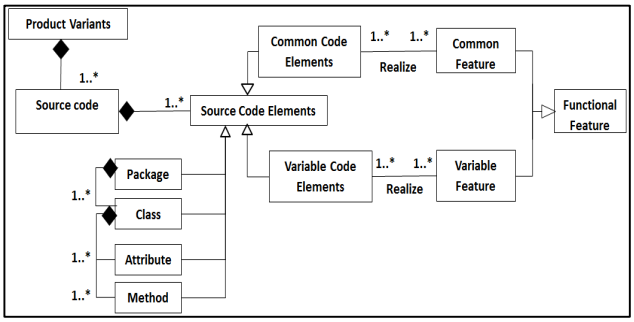

Figure1. Feature-Source code mapping model in product variants.

underlying intuition behind this goal is to map less features to less implementation in order to apply LSI in efficient way. In our previous work published in [22], we deal with product variants as a collection and divided this collection at feature and implementation levels into two partitions (common and variable partitions). At feature level, common and variable partitions represent common and optional features across product variants respectively. At implementation, common and variable partitions implement common and optional features respectively. During this current work, we will divide each product variant at feature and implementation levels into two partitions considering variability and commonality distribution across product variants.

Presence or absence a feature in product variant should be reflected in the implementation by presence or absence corresponding source code elements. Thus, we proposed an approach to divide each product variant at feature and implantation levels into two partitions as follow:

1) At Feature Level

We rely on lexical similarity of feature names and descriptions to determine common features across product variants such as core features in our illustrative example (*open, save and create*).

Algorithm1: ICFeatures.

Input:features sets of product variants PVF = { PF_1,…,PF_n}.
Output:common features (F_{sn}).

1: $F_{sn} := PF_1$
2: **for** i := 2 **to** length [PVF] **do**
3: $F_{sn}:=F_{sn} \cap PF_i$
4: **return** F_{sn}

For a set of features of set of product variants, our approach firstly defines a subset of same name features (F_{sn}) according to given above *ICFeatures* algorithm. *ICFeatures* takes as input sets of features of product variants (*PVF*) and return common features (F_{sn}). *PVF* represents a multiset data structure where each set corresponds to specific product variant. For instance, PF_1 corresponds to product variant1 features. Step three compute shared features by conducting an intersection among all product variants features.

A feature may be renamed to response changes in software environment or the adoption of different technology [2]. Our approach considers this issue into account by computing lexical similarity pair-wisely for those features that don't have same name based on longest common subsequence (LCS) of their feature descriptions [21]. For example, for two features f_1 and f_2 where $f_1 \in PF_1$, $f_2 \in PF_2$ and f_1.name $\neq f_2$.name, if LCS for description of f_1 and f_2 has the same subsequence terms we can consider f_1 and f_2 represent the same feature.

By identifying common features across product variants, the rest features in any variant represent optional features. In our illustrative example, *core* features are common features across product variant while *search* and *edit* features represent optional features in *T_Editor_V1.1*.

2) At Implementation Level

Our approach analyzes source code of a set of product variants itself. Source code for each product variant is decomposed into a set of elementary construction units (ECU). ECU considers packages and classes. ECU takes the following format:

ECU = { Package.Name_Class.Name}.

This representation is inspired by the model construction operations proposed by [20]. Each product variant P_i is encoded as a set of ECUs, i.e. $P_i=\{ECU_1,ECU_2,\ldots,ECU_n\}$. We can note that our ECU can appear any structural changes at package and class levels. We call these changes as variations levels in the source code. These variations can reflect any changes at feature level (e.g. add or remove features) directly in the implementation level by adding or removing corresponding OOBEs.

Algorithm2: *IC_ECUs*

Input: Set of product variants (AllPV) abstracted as ECU. AllPV = $\{P_1,P_2,\ldots,P_n\}$.
Output: Common ECUs (C_ECUs)

1: C_ECUs := P_1
2: **for** i := 2 **to** length [AllPV] **do**
3: C_ECUs:=C_ECUs ∩ P_i
4: **return** C_ECUs

In order to identify common *ECUs* shared by all product variants, we proposed above *IC_ECUs* algorithm. *IC_ECUs* takes as input a set of product variants abstracted as a set of *ECUs* and returns common *ECUs* (common source code elements) across product variants. Step three compute shared *ECUs* by conducting an intersection among all *ECUs* of a set of product variants. These shared *ECUs* implement common features.

By identifying common *ECUs* (C_ECUs) across product variants, the rest ECUs in any variant represent variable *ECUs* (V_ ECUs). V_ ECUs represents variable source code elements. In our illustrative example, if we consider that *T_Editor_V1.0* is implemented by $\{ECU_1, ECU_2\}$ and *T_Editor_V1.1* is realized by $\{ECU_1, ECU_2, ECU_3, ECU_4, ECU_5\}$. Our approach reports that C_ *ECUs* = $\{ECU_1, ECU_2\}$ while *V_ECU* = $\{ECU_3, ECU_4, ECU5\}$.

E. Recovering Traceability Links By LSI

Domain knowledge and concepts are recorded in the source code through identifiers. Thus, our approach uses LSI for analyzing these elements to identify traceability links between common features and common source code elements (classes), and between optional features and variable source code elements in each product variant. Our applying of LSI is similar to [1]. It involves building LSI corpus and queries.

1) Building LSI Corpus

Our approach depends on four steps to process source code: (1) Identifiers extraction. (2) Tokenization. (3) Tokens manipulation. (4) Determining document granularity.

Firstly, identifiers extraction needs a parser to extract all source code information. During our work we used a Java parser to build abstract syntax tree of the source code that can be queried to extract required identifiers.

Secondly, identifiers must be tokenized. We considered two commonly styles for identifiers: one is the combination of words using underscore as delimiters (e.g., traceability_links); and the other is the combination of tokens using letter capitalization for separation (e.g., TraceabilityLinks). All identifiers that follow these rules are tokenized into singular tokens (e.g., traceability links for the above examples).

Thirdly, tokens are manipulated by reducing every token to its root. For example, *take, took* and *taken* are reduced to the same root *take*. Finally, we choose each class to be a separate document. A document contains lines of all identifier inside a class.

After source code processing, each product variant (*P*) is decomposed into a set of documents. These documents represent LSI corpus.

2) Building Queries

In our approach, LSI uses feature name and description as query to retrieve classes relevant to a specific feature. Our approach creates a document for each feature. This document contains feature name description, and it also is manipulated likes source code. Our approach extracts tokens from feature name and description. It uses white space and punctuation marks as delimiters. Then it reduces every token to its root.

3) Establishing traceability links

We feed LSI with documents and queries to build topic model. LSI builds a vector of weights for each document and query. Each weight represents a probability of affiliation for a given document and a query to the same topic. Then, LSI measures the similarity between queries and documents using cosine similarity. It returns a list of documents ordered based on their similarity against each

query. In our work, we consider the most widely used threshold for cosine similarity that equals to 0.70 [1], i.e., documents that will be retrieved have a similarity with a query greater than or equal to the threshold value.

IV. EXPERIMENTAL RESULTS AND EVALUATION

In this section, we show the case study used for the evaluation of our approach, present the evaluation metrics and discuss the experimental results.

A. Case Study

We have applied our approach on a set of product variants from ArgoUML modelling tool. These variants represent members of ArgoUML-SPL[1] published in [10]. We generated four variants from ArgoUML-SPL as shown in table 2 below.

The four variants provide two common features (class diagram and cognitive support features) and seven optional features (state, collaboration, usecase, activity, deployment and sequence diagram features). The advantage of ArgoUML-SPL is that it implements features at different levels where features are implemented at package, class, method and attribute levels. Preprocessor directives have been used to annotate the source code elements associated to each feature. This pre-compilation process allows us to establish the truth links (real implementation for each feature) to evaluate the effectiveness of our approach.

Table 2: Set of ArgoUML-SPL members.

Products	Features
Product1	Class, cognitive, sequence, usecase, state, activity.
Product2	Class, cognitive, sequence, usecase, collaboration, activity.
Product3	Class, cognitive, collaboration, deployment, state.
Product4	Class, cognitive, state, activity, collaboration, deployment.

B. Evaluation Measures

We have used two measures to evaluate our approach: *Precision and Recall.* These measures are commonly used to evaluate IR methods [1].

1) Precision

Precision describes the precision of retrieved traceability links for a given feature. Precision is the percentage of correctly retrieved links (classes) to the total number of retrieved links. Equation 1 below represents *precision* metric equation where *i* ranges over the entire features set.

$$\text{Precision} = \frac{\sum_i \text{Correctly Retrieved Links}}{\sum_i \text{Total Retrieved Links}} \% \qquad EQ(1)$$

Precision values can have any value in the interval [0, 1]. Higher precision values mean better results for the approach that establishes traceability links.

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

2) Recall

Recall quantifies number of relevant links that are retrieved for a given feature. Recall is the percentage of correctly retrieved links to the total number of relevant links. Below given equation 2 represents *recall* metric equation where *i* ranges over the entire features set.

$$\text{Recall} = \frac{\sum_i \text{Correctly Retrieved Links}}{\sum_i \text{Total Relevant Links}} \% \qquad EQ(2)$$

Recall values can have any value in the interval [0, 1]. Higher recall values mean better results for the approach that establishes traceability links.

C. Performance of Our Approach

LSI associate related tokens into topics based on their occurrences in the documents in a corpus. The most important parameter to LSI is the number of topics that should be used for topic-model building. We need enough topics to catch real term relations. Too many topics lead to associate irrelevant terms. Small number of topics lead to lost relevant terms. According to Dumais et al. [23], the number of topics is between 235 and 250 for natural language. For a corpus of source code files, Poshyvanyk et al. [24] recommended that the number of topics is 750.

In this work we cannot use a fixed number of topics for LSI because we have different size of partitions. Thus, we use a factor *k* between 0.01 and 0.04 to determine number of topics. The number of topics (#topics) = $k \times doc_d$, where doc_d is document dimensionality of term-document matrix that is generated by LSI. We evaluate the performance of our approach for #topic at *k= 0.01, 0.02, 0.03 and 0.04.*

Figures 2 and 3 compare the precision and recall results for our approach against applying LSI in conventional way. The graphs *A,B,C,D* given in figures 2 and 3 corresponds to *Product1,Product2,Product3 and Product4* respectively. The X-axis in the graphs represents the number of LSI topics while Y-axis in the figures 2 and 3 correspond to precision and recall respectively. It can be noticed that our approach always gives a better precision and recall results than applying LSI in conventional way.

The threat to the validity of our approach is that if developers don't use the same vocabularies to name source code identifiers across product variants. This would means that lexical matching at implementation level will be effected. However, when a company has to develop a new product that is similar, but not identical, to existing ones, an existing product is cloned and later modified according to new demands.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented an approach that uses LSI in effective way to establish traceability links between the object oriented source code of a collection of product variants and a given features of these variants. Our approach exploits variability and commonality distribution of product

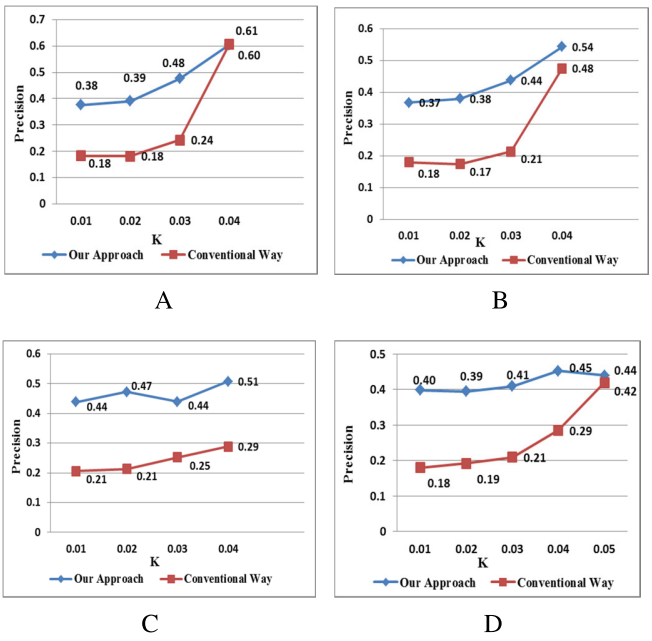

Figure 2. Precision results for our approach against applying LSI in conventional way.

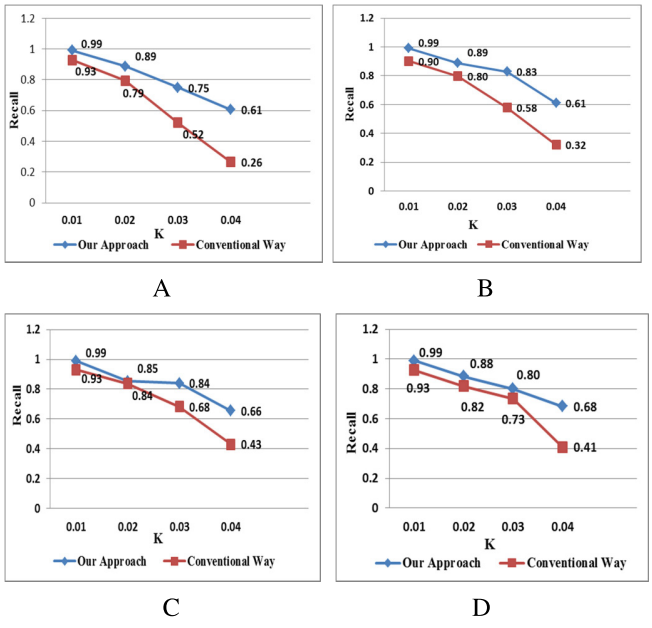

Figure 3.Recall results for our approach against applying LSI in conventional way.

variants to reduce features and implementation spaces such that LSI can be applied in efficient way. The evaluation of our approach with a collection of four ArgoUML-SPL products showed that our approach significantly outperforms applying LSI in conventional way according to the precision and recall metrics.

In our future work, we plan to use existing relationships between source code elements (e.g., method call, class inheritance and son on) to improve the relevance of identified traceability links. This will require a definition for semantic similarity measure between source code elements.

REFERENCES

[1]. A. Marcus and J.I. Maletic. Recovering documentation-to-source code traceability links using Latent Semantic Indexing. ICSE 2003, pp.125-137.

[2]. Y. Xue, Z. Xing, and S. Jarzabek. Understanding feature evolution in a family of product variants. WCRE 2010, pp.109-118.

[3]. P.Clements and L.Northrop. Software product lines: practices patterns. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2001

[4]. Y. Ghanam and F.Maurer. Linking feature models to code artifacts using executable acceptance tests. In Proceedings of the 14th international conference on Software product lines: going beyond (SPLC'10), Jan Bosch and Jaejoon Lee (Eds.). Springer-Verlag, Berlin, Heidelberg,2010, 211-225.

[5]. P.Ye, X.Peng, Y.Xue and S. Jarzabek.: A Case Study of Variation Mechanism in an Industrial Product Line. ICSR. 2009,126-136.

[6]. D.Andrea, F.Fausto, O.Rocco and T.Genoveffa. Recovering traceability links in software artifact management systems using information retrieval methods. ACM Trans. Softw. Eng. Methodol. 16, 4,20007, Article 13 .

[7]. D.Beuche, H.Papajewski, S.Wolfgang.. Variability management with feature models. Sci. Comput. Program. 53, 3 (December 2004), 333-352. 352.

[8]. I. John and M. Eisenbarth, "A decade of scoping: a survey," in SPLC, ser. ACM International Conference Proceeding Series, D. Muthig and J. D. McGregor, Eds., vol. 446. ACM, 2009, pp. 31–40.

[9]. B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey, JSME, 28 Nov, 2011.

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

[11]. S. Wang, D. Lo, Z. Xing, and L. Jiang. Concern localization using information retrieval: An empirical study on Linux kernel. WCRE 2011, pp. 92-96.

[12]. J. Rubin and M. Chechik. Locating distinguishing features using diff sets. InProceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering(ASE 2012). ACM, New York, NY, USA, 242-245.

[13]. G. Orlena and F. Anthony. An analysis of the requirements traceability problem. In Proceedings of 1st International Conference on Requirements Engineering (Colorado Springs, CO). IEEE Computer Society Press,1994, Los Alamitos, CA, 94–101.

[14]. S. George and Z. Andrea. Software Traceability: A Roadmap, in Handbook of Software Engineering and Knowledge Engineering, Chang, S. K., Ed. World Scientific Publishing Co,2004 , pp. 395-428.

[15]. W.Shaowei, L.David, X.Zhenchang and J.Lingxiao .Concern Localization using Information Retrieval: An Empirical Study on Linux Kernel. In Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE '11). IEEE Computer Society, Washington, DC, USA, 92-96.

[16]. S.Gerard and M.Michael. Introduction to Modern Information Retrieval. McGraw-Hill, Inc.,1996, New York, NY, USA.

[17]. M.Andrian and M.Jonathan. "Identification of High Level Concept Clones in Source Code", in Proceedings of Automated Software Engineering (ASE'01), San Diego, CA, November 26-29 2001, pp. 107-114.

[18]. M. Riebisch, “Towards a more precise definition of feature models,” in Modelling Variability for Object-Oriented Prod- uct Lines, M. Riebisch, J. O. Coplien, and D. Streitferdt, Eds. Norderstedt: BookOnDemand Publ. Co, 2003, pp. 64–76.

[19]. K. Kyo, C.Cohen,H.James,N.William and P.Spencer.Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990,Carnegie Mellon University.

[20]. X. Blanc, I. Mounier, A. Mougenot, and T. Mens, “Detecting model inconsistency through operation-based model construction,” in ICSE, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds.ACM, 2008, pp. 511–520.

[21]. L. Bergroth and H. Hakonen and T. Raita. A Survey of Longest Common Subsequence Algorithms. SPIRE 2000, pp. 39–48.

[22]. E.Hamzeh, S.Abdelhak-Djamal, D.Christophe and A.Ra'Fat. Recovering Traceability links between Feature Models and Source Code of Product Variants. ACM VARY Workshop (VARY: VARiability for You @ MODELS 2012, Sept. 30th - Oct. 5th, 2012 - Innsbruck, Austria.

[23]. S.T. Dumais. LSI meets TREC: A status report, in Proceeding of Text Retrieval Conference, pp. 137-152. 1992.

[24]. D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Guéhéneuc, and G. Antoniol. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. ICPC, pp. 137-148, 2006.