ACADÉMIE DE MONTPELLIER

UNIVERSITÉ MONTPELLIER II

— **S**CIENCES ET **T**ECHNIQUES DU **L**ANGUEDOC —

THÈSE

présentée au Laboratoire d'Informatique de Robotique
et de Microélectronique de Montpellier pour
obtenir le diplôme de doctorat

SPÉCIALITÉ	:	**INFORMATIQUE**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

Recovering Traceability Links between Artifacts of Software Variants in the Context of Software Product Line Engineering

par

Hamzeh Eyal Salman

Soutenue le 17 Juin 2014, devant le jury composé de :

Jean-Claude ROYER, Professeur, École des Mines de Nantes ..... Rapporteur
Henri BASSON, Professeur, University of Lille-Littoral ..... Rapporteur
Mourad OUSSALAH, Professeur, University of Nantes, ... Examinateur
Marianne HUCHARD, Professeur, LIRMM, Université Montpellier II ..... Examinateur
Christophe DONY, Professeur, LIRMM, Université Montpellier II ..... Directeur de Thèse
Abdelhak-Djamel SERIAI, Maitre de Conférences, LIRMM, Université Montpellier II .... Co-Directeur de Thèse

Contents

Acknowledgment

Saying thank you is more than good manners. It is good spirituality.

Alfred PAINTER.

First of all, I would praise Allah, the Grand Creator, who moves out the brains and hearts of mankind from the darkness of ignorance into the light of science and bestows his good blessings upon us to be always thankful and grateful.

A major research work like this is never the work of anyone alone. The contributions of many different people, in their different ways, have made this possible. I would like to extend my appreciation especially to the following:

Foremost, I would like to express my sincere gratitude to my supervisor *Dr. Abdelhak-Djamel Seriai* for his continuous support, patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my Ph.D study. I cannot remember how many meetings I have got. I am not sure whether a PhD student already had the same amount of effort from his supervisor. I will never be able to truly express my appreciation. I can only say: Thanks for the support and concern.

Besides *Dr. Seriai*, I would like to thank Director of the thesis *Prof. Christophe Dony* for his patience, encouragement, humorous spirit and insightful comments. He has cheerfully answered my queries, learned me almost everything about research, about speaking and writing, checked my examples, assisted me in a many ways with this thesis. Thanks as many as my heart's beats.

My sincere thanks and great respect also go to *MaREL* team, especially, *Prof. Huchard*. The person who amazed me with his humbleness, good manners, kind assistance and valuable advice. Thanks for the great and splendid efforts you did.

I would like to thank the members of my thesis committee. Thank you for having accepted to review my thesis and for their time in reading and commenting on my thesis. This jury composed of many researchers that have inspired me a lot. When your research is related to recovering traceability links between software artifacts, and software product line engineering, you must read and study carefully the prominent works of *Prof. Jean-Claude Royer*. Prof.*Henri Basson* had also a direct impact on my research regarding his major contributions in the area of change impact analysis. When your

research is also related to software architecture, you must study carefully the prominent works of *Prof. Mourad Oussalah*. *Prof. Marianne Huchard* had also a direct impact on my research regarding his major contributions in the area of formal concept analysis.

I would like to thank with endless esteem to my dear family who paved the way of knowledge to me and opened my eyes to uncover the world. Thanks for their patience, encouragement and support which made me achieve something considered as a great pride for them.

Finally, I am truely and deeply indebted to ERASMUS MUNDUS staff at University of Montpellier 2 for their generous financial support. A big thank to them. I will never forget the best friends ever known, Hajer, Seza, Sahar, Jamal, Ayman, Mohammed, Noor, Anas, Zakeria, Abd-Rahman, Tareq who were helpful and consolidated with me. For all, I say thank you!

Abstract

Software Product Line Engineering (SPLE) is a software engineering discipline providing methods to promote systematic software reuse for developing short time-to-market and quality products in a cost-efficient way. SPLE leverages what Software Product Line (SPL) members have in common and manages what varies among them. The idea behind SPLE is to build core assets consisting of all reusable software artifacts (such as requirements, architecture, components, etc.) that can be leveraged to develop SPL's products in a prescribed way. Creating these core assets is driven by features provided by SPL products. Unfortunately, building SPL core assets from scratch is a costly task and requires a long time, which leads to increase time-to-market and up-front investment. To reduce these costs, existing similar product variants developed by ad-hoc reuse should be re-engineered to build SPLs. In this context, our thesis proposes three contributions.

Firstly, we proposed an approach to recover traceability links between features and their implementing source code in a collection of product variants. This helps to understand source code of product variants and facilitate new product derivation from SPL's core assets. The proposed approach is based on Information Retrieval (IR) for recovering such traceability links. In our experimental evaluation, we showed that our approach outperforms the conventional application of IR as well as the most recent and relevant work on the subject. Secondly, we proposed an approach, based on traceability links recovered in the first contribution, to study feature-level Change Impact Analysis (CIA) for changes made to source code of features of product variants. This approach helps to conduct change management from a SPL manager's point of view. This allows him to decide which change strategy should be executed, as there is often more than one change that can solve the same problem. In our experimental evaluation, we proved the effectiveness of our approach in terms of the most widely used metrics on the subject. Finally, based on traceability recovered in the first contribution, we proposed an approach to contribute towards building Software Product Line Architecture (SPLA) and linking its elements with features. Our focus is to identify mandatory components and variation points of components. Therefore, we proposed a set of algorithms to identify this commonality and variability across a given collection of product variants. According to the experimental evaluation, the efficiency of these algorithms mainly depends on the available product configurations.

Keywords: *Software product line engineering, product variants, reuse, variability, feature location, traceability, architecture, change impact analysis, source code, re-engineering, formal concept analysis, information retrieval, algorithms.*

Résumé

L'ingénierie des lignes de produits logiciels (Software Product Line Engineering-SPLE en Anglais) est une discipline qui met en œuvre des principes de réutilisation pour le développement efficace de familles de produits. Une famille de produits logiciels est un ensemble de logiciels similaires, ayant des fonctionnalités communes, mais néanmoins différents selon divers aspects; nous parlerons des différentes variantes d'un logiciel. L'utilisation d'une ligne de produit permet de développer les nouveaux produits d'une famille plus vite et d'augmenter la qualité de chacun d'eux. Ces avantages sont liés au fait que les éléments communs aux membres d'une même famille (besoin, architecture, code source, etc.) sont réutilisés et adaptés. Créer de toutes pièces une ligne de produits est une tâche difficile, coûteuse et longue. L'idée sous-jacente à ce travail est qu'une ligne de produits peut être créée par la ré-ingénierie de logiciels similaires (de la même famille) existants, qui ont été préalablement développés de manière ad-hoc. Dans ce contexte, la contribution de cette thèse est triple.

La première contribution est la proposition d'une approche pour l'identification des liens de traçabilité entre les caractéristiques (features) d'une application et les parties du code source qui les implémentent, et ce pour toutes les variantes d'une application. Ces liens sont utiles pour générer (dériver) de nouveaux logiciels par la sélection de leurs caractéristiques. L'approche proposée est principalement basée sur l'amélioration de la technique conventionnelle de recherche d'information (Information Retrieval –IR en Anglais) et des approches les plus récentes dans ce domaine. Cette amélioration est liée à deux facteurs. Le premier facteur est l'exploitation des informations liées aux éléments communs ou variables des caractéristiques et du code source des produits logiciels analysés. Le deuxième facteur concerne l'exploitation des similarités et des dépendances entre les éléments du code source. Les résultats que nous avons obtenus par expérimentation confirment l'efficacité de notre approche. Dans la deuxième contribution, nous appliquons nos résultats précédents (contribution no 1) à l'analyse d'impact (Change Impact Analysis –CIA en Anglais). Nous proposons un algorithme permettant à un gestionnaire de ligne de produit ou de produit de détecter quelles les caractéristiques (choix de configuration du logiciel) impactées par une modification du code. Cet algorithme améliore les résultats les plus récents dans ce domaine en permettant de mesurer à quel degré la réalisation d'une caractéristique est impactée par une modification. Dans la troisième contribution nous exploitons à nouveau ces liens de traçabilité (contribution No 1) pour proposer une approche permettant de satisfaire deux objectifs. Le premier concerne l'extraction de l'architecture de la ligne de produits. Nous proposons un ensemble d'algorithmes pour identifier les points de variabilité architecturale à travers l'identification des points de variabilité au niveau des caractéristiques. Le deuxième objectif concerne l'identification des liens de traçabilité entre les caractéristiques et les éléments de l'architecture de la ligne de produits. Les résultats de l'expérimentation

montre que l'efficacité de notre approche dépend de l'ensemble des configurations de caractéristiques utilisé (disponibles via les variantes de produits analysés).

Mots-clés : Traçabilité, localisation de caractéristiques, recherche d'information, variantes de produits logiciels, variabilité, Ingénierie des lignes de produits, Analyse Formelle de concepts, architecture, analyse de l'impact du changement, algorithmes, réutilisation, logiciels similaires, réingénierie, code source.

List of Acronyms

SPLE	Software Product Line Engineering
SPL	Software Product Line
SPLA	Software Product Line Architecture
FM	Feature Model
SVD	Singular Value Decomposition
LSI	Latent Semantic Indexing
VSM	Vector Space Model
FCA	Formal Concept Analysis
GSH	Galois Sub-Hierarchy Lattice
AHC	Agglomerative Hierarchical Clustering
CIA	Change Impact Analysis
EIS	Estimated Impact Set
AIS	Actual Impact Set
CSC	Change Set of Classes
ISC	Impact Set of Classes
IDM	Impact Degree Metric
CAM	Changeability Assessment Metric
VP	Variation point
PC	Product Configuration

CHAPTER 1

Introduction

1.1 Research Context: Software Product Line Engineering and Product Variants

SPLE is a software engineering discipline providing methods to promote systematic software reuse for developing a family of software products, rather than individual ones [Clements et Northrop, 2001]. This family is known as *Software Product Line (SPL)*. SPLE focuses on efficiently producing and maintaining multiple similar software products, leveraging what they have in common and managing what varies among them. This is analogous to what is performed in the automotive industry, where the focus is on creating a single production line, which consists of a family of customized cars with common characteristics. In SPLE, SPL's products of a domain are developed from common and reusable software artifacts, called core assets. An *asset* is any reusable software artifact that can be employed in the development of a software product [Pohl *et al.*, 2010]. Core assets can be, but are not limited to, requirement documents, architecture, components, source code, test cases, etc. The basic assumption behind SPLE is that reuse a large scale works best in families of related systems (i.e., SPLs) developed from the same core assets which have been initially identified and co-developed.

SPLE has economic considerations, which leads companies to transit their software development strategy toward SPLE. It allows companies to produce a set of related products at lower costs, in a shorter time and with higher quality [Pohl *et al.*, 2010][Linden *et al.*, 2007][Royer et Arboleda, 2012]. SPLE, of course, reduces the cost of software products development because core assets are reused in several products, this implies a cost reduction for each product and short time to market. In SPLE, the core assets are tested in many products, this means providing a high chance to detect fault and errors in the course of time. This leads to increase the quality of all generated products.

There are two essential tasks playing important roles for successful SPLE: *variability management*

and *product derivation* [Clements et Northrop, 2001]. Firstly, variability, in the context of SPLE, can be understood as differences among SPL's products in terms of provided features. A *feature* is defined as "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems" [Kang *et al.*, 1990]. A feature can be seen as a bundle of requirements abstracted at the feature level as a feature. Moreover, building core assets is driven by features that SPL products should provide. The concept of *commonality* is highly correlated to variability. It refers to features that are part of each product in SPL (i.e., mandatory features). Leveraging commonality and managing variability allow the provision of a range of miscellaneous products (SPL) which meet the different and individual needs of customers in a particular domain. Commonality and variability in SPLE are managed by *Feature Models*, a formalism introduced by [Kang *et al.*, 1990] and now widely used in SPLE. Secondly, product derivation is the process of generating a product from core assets [Thao *et al.*, 2008]. This process determines which assets should be selected according to features selection, and specifies how those assets are composed in order to build the desired product [Deelstra *et al.*, 2004]. Features selection is driven by customer needs. This process is automatically performed for generating many products in shorter time, which lead to meet the growing needs of customers in a particular domain.

Traceability links between artifacts of SPL's core assets play a pivot role for automating product derivation. The term traceability has different meanings in different contexts. In the context of SPL's core assets, traceability is the ability to relate the different artifacts of core assets created in the development life cycle with one another [Ajila et Kaba, 2004]. Such traceability links relate features with core assets that implement those features, which facilitate and automate the product derivation process [Shen *et al.*, 2009][bin Abid, 2009]. On the other hand, traceability in SPLE is considered as an important element to manage the complexity of variability by ensuring the correct binding of variability throughout all derived products and by highlighting variability throughout all SPL's core assets [Pohl *et al.*, 2010]. Additionally, traceability links are essential for maintenance tasks, such as change impact analysis. When a change is induced, the impact of change is not limited to a certain type of software artifacts, but it goes through all software artifacts of different levels of abstraction [De Lucia *et al.*, 2008]. Therefore, traceability is needed to detect change propagation between different types of artifacts.

Highlighting variability at architectural level is the basis for building Software Product Line Architecture (SPLA). SPLA is a key asset in SPL's core assets. Thus, SPLE is termed *architecture-centric* [Linden *et al.*, 2007]. SPLA is a core architecture that captures the high level design for the products of the SPL, including commonality and variability documented in the variability model (or feature model) [Pohl *et al.*, 2010]. Variability in terms of features can be reflected in SPLA as variation points of components. Such a variation point represents a location in the SPLA to switch between alternative component (s) in order to derive specific architecture for each product in SPL. Selection of the appropriate components depends on the chosen features. For this, traceability links between feature and SPLA is needed to bind variability at architectural level, and hence deriving architecture for each member of SPL.

It is common for companies developing variants of a software product to accommodate different customer needs. These variants provide common features (mandatory) but they also differ from one another by providing unique features or feature combinations. These products are known *product variants* [Yinxing *et al.*, 2010]. Often, product variants evolve from an initial product developed for and successfully used by the first customer. Unfortunately, product variants are not developed and

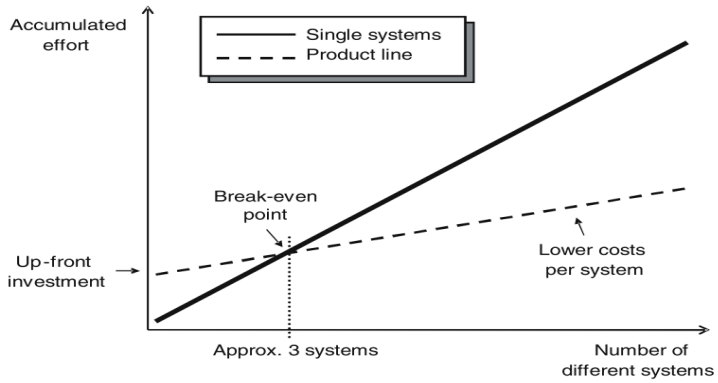

Figure 1.1 : Accumulated Costs for SPL and Independently Systems [Pohl *et al.*, 2010]

maintained as SPL. They are often developed by using ad-hoc reuse approaches such as "clone-and-own". Companies develop separate product variants where a new product is built by ad-hoc copying and modification of various existing variants to fit new purposes. Separately maintaining product variants causes challenges. Changes in the code of common features must be repeated across product variants. Moreover, software engineers must know how the feature (i.e., its implementation) is supposed to interact (or not interact) with other features at source code level in every product variant independently [Calder *et al.*, 2003]. As the number of features and the number of product variants grow, developing product variants in such a way becomes more challenging and costly.

1.2 Problem Statement

As mentioned earlier, SPL's core assets play a pivotal role in SPLE. However, building SPL's core assets from scratch is a costly task and requires a long time, which leads to increase time-to-market [Pohl *et al.*, 2010]. Figure 1.1 shows the accumulated costs required to develop n software products in both SPLE and traditional way of software development (one software system at a time). The solid line represents the costs of developing software products independently, while the dashed line represents the development costs using SPLE. In the case of the development of a few products, the costs of SPLE are high. This is because the SPL's core assets are built from scratch to generate not only small number of products but also to support generation of the full scope of products required for the foreseeable horizon. Thus, this leads to increase up-front investment. However, these costs are significantly reduced for the development of many products. The location at which both curves intersect represents the break-even point. At this point, the costs are the same for developing the systems independently as for developing them using SPLE.

Generally, companies cannot afford to start developing a SPL from scratch, and hence they have to reuse as much existing software assets of product variants as possible. The existing assets represent a starting point for building SPL's assets. In product variants, features and source code are available assets while SPLA can be reverse-engineered from these variants. Features are available due to the need for product customization, as product variants are a series of customized products to meet specific needs of different customers. These needs can be features provided by these variants. Source

code is always available. These assets (features, source code and SPLA) should be linked together to be a part of SPL's core assets, as core assets in SPLE are linked.

The global goal of this thesis is to support re-engineering product variants into SPLs. In this goal, we address the following problems.

1. **Finding traceability links between features and their implementing source code elements in a collection of product variants**

Traceability links between features and their implementing source code elements are necessary to understand source code of product variants, and then reuse right features (resp. their implementations) for developing new products taking advantages of SPLE [Lucia *et al.*, 2007]. Such traceability is also essential for facilitating and automating new product derivation from SPL's core assets, when the re-engineering process is completed. Figure 1.2 shows an example of traceability links in a collection of two product variants. Each product has specific features which are only chosen in that product and also these products have shared features. The links between these features and classes represent traceability links that determine which features are implemented by which classes. In the literature, the process of identifying traceability links between features and their implementing source code elements is known as *feature location* [Biggerstaff *et al.*, 1993][Bogdan *et al.*, 2013]. Throughout the thesis, the term of *feature implementation* refers to the source code elements that implement a feature.

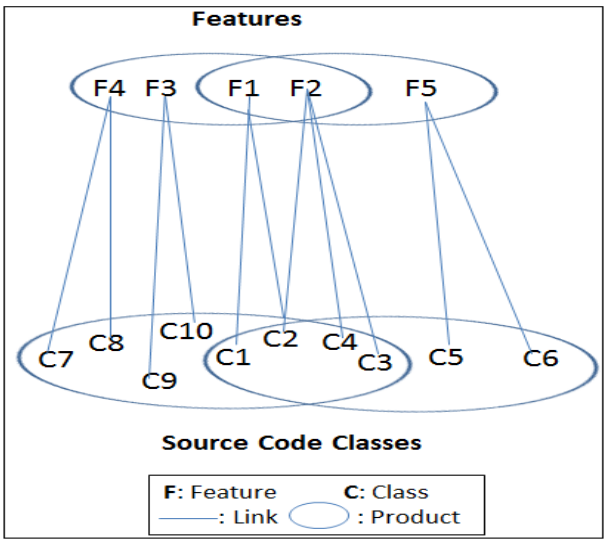

Figure 1.2 : An Example of Traceability Links in a Collection of Two Product Variants.

Then, we study how to use feature location for addressing the following problems concerned with our global goal.

2. **Performing change impact analysis at feature level for changes made to source code**

Features (resp. their implementing source code elements) obtained from product variants may need to be changed for adapting SPLE context by adding or removing requirements (resp. their source code

elements) to meet new demands of customers [Liu *et al.*, 2006]. The change may impact the implementation of 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 determine feature(s) that may be impact for a given change proposal before the change is implemented. 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, and hence which resources should be reserved. As a feature is an agreement among all stakeholders (including managers) about what systems should do, feature-level CIA allows managers to deeply understand the impact of a given change proposal, rather than technical details. Furthermore, such impact analysis detect the introduction of undesirable interactions between feature implementations. Feature-level CIA represents a maintenance task for features (resp. their implementations) obtained from product variants. The role of feature location in this task is to determine a feature's implementation need to be changed and help the CIA process to determine affected features due to source code changes.

3. **Supporting reverse-engineering SPLA from product variants and establishing feature-to-architecture traceability links**

Developing SPLA from scratch is a costly task because it represents an infrastructure to derive components not only for one architecture but also for all architectures of SPL's products. As a result, existing product variants should be reused as much as possible to build SPLA by extracting components and organizing these components as mandatory and as members of variation points. This organization represents an important step toward reverse-engineering SPLA from product variants. The role of feature location in this task is to help extract components only from the implementation of features. Also, it is used to highlight variability at architectural level, in order to identify variation points of components by establishing traceability links between features and components, as we will see later.

1.3 Limitations of the Existing Approaches

Information Retrieval (IR) techniques are used widely for locating feature implementations. Most IR-based feature location approaches conduct a textual matching between all features (i.e., their feature descriptions) for a single software product and entire source code information of that product. These features and the associated source code are called IR search spaces. These approaches deal with product variants as singular independent entities. However, by considering product variants as a family of similar and related products, an additional input to the process of IR-based feature location is obtained. This input represents commonality and variability across product variants, which leads to reducing IR search spaces and hence improves the results of IR-feature location process. There are a few works that consider commonality and variability of product variants. These works do not achieve minimal reduction of IR search spaces and also do not pay attention to features information that represents a guide to IR for locating feature implementations. Moreover, none of the surveyed approaches consider the abstraction gap between feature and source code levels, which may hinder the IR-feature location process.

CIA at feature level for source code changes is seldom considered. Most surveyed approaches perform change impact at the source code level with few works completed at requirement and design levels. There is only one work that studies the change impact at the feature level for source code changes. However, this work does not support the purposes of SPL's manager concerned with change management, as it does not provide quantitative metrics to help with making decisions.

Although SPLA is a key asset in SPL's core assets and its development from scratch is a costly task, none of existing approaches studied how to support reverse engineering SPLA from product variants. All existing works support building SPLA from scratch, i.e., forward engineering way to build SPLA in the SPLE context.

1.4 Contributions

During this thesis, we make three contributions as follows:

- ***Feature location in a collection of product variants***
  We propose an approach to support feature location in a collection of product variants based on Information Retrieval (IR) techniques namely, Latent Semantic Indexing (LSI). This approach improves the performance of IR-based feature location in terms of the most used metrics in the subject: *precision, recall* and *F-measure*. This improvement is based on two kinds of reductions. Firstly, exploiting commonality and variability across product variants to reduce feature and source code spaces where IR is applied. Secondly, reducing abstraction gap between feature and source code levels by introducing *code-topic* as an intermediate level. We use Formal Concept Analysis (FCA) to reduce IR spaces by analyzing commonality and variability across product variants in order to obtain minimal disjoint sets of features (resp. their implementing source code elements). We also investigate the results of Agglomerative Hierarchical Clustering (AHC) and FCA to identify *code-topics*.

- ***Change impact analysis at the feature level: based on feature location***
  In the second contribution, we propose an approach to support feature-level CIA for source code changes made to the implementation of features obtained from product variants. This approach takes, as input, a set of classes to be changed and feature implementations. It returns, as output, a ranked list of affected features. We propose two metrics adapted to feature-level CIA: impact degree and changeability assessment metrics. The former is used to measure to which degree a specific feature can be affected by a given change proposal. The latter is used to measure the percentage of features that will be impacted by a given change proposal.

- ***Toward reverse engineering SPLA from product variants and establishing feature-to-architecture traceability links: based on feature location***
  In the third contribution, we propose an approach to support reverse-engineering of SPLA from product variants. As SPLA encompasses commonality and variability at feature level, we first propose determining mandatory features (commonality) and variation point of features (variability) across product variants. Then, we highlight this commonality and variability at the architectural level by extracting components from feature implementations and then establishing traceability links between features and extracted components. This highlighting allows

identification of mandatory components and variation points of components as an important step towards building SPLA.

1.5 Thesis Organization

The contents of the remaining chapters are as follows:

1.5.1 Part I: Background

- **Chapter** 2 presents a base of knowledge that is used as a common language to explain our approaches throughout this thesis. Firstly, main concepts and definitions about SPLE are given. Then, we give a necessary background about IR techniques with illustrative example. Next, we introduce FCA and their definition with illustrative example. Finally, we detail case studies used through the thesis.

1.5.2 Part II: State-Of-The-Art

- **Chapter** 3 reviews the state-of-the-art literature about IR-based feature location. Firstly, we present the types of traceability links and their semantics. Then, we give a general classification of feature location approaches. Next, we focus on IR-based feature location approaches. Finally, we point out advantages and disadvantages of studied approaches.

- **Chapter** 4 reviews the state-of-the-art literature about the application of feature location for feature-level CIA and reverse engineering SPLA from product variants. We present the main concepts in CIA and SPLA and study approaches related to feature-level CIA and reverse-engineering SPLA. Finally, we evaluate the studied approaches.

1.5.3 Part III: Contributions

- **Chapter** 5 reports our contribution for supporting feature location in a collection of product variants. Firstly, we introduce the basic assumptions considered. Then, we explain how to improve the performance of IR-based feature location in a collection of product variants through reducing IR search spaces and reducing the abstraction gap between feature and source code levels. Then, we present how to locate features using LSI. Finally, we present our experimental validation.

- **Chapter** 6 presents our approach that supports feature-level CIA for source code changes. Firstly, we determine the impacted set of classes due to source code changes. Next, we use FCA to analyze feature coupling relations. Then, we present how to query the concept lattice generated by FCA to compute a ranked list of affected features. Finally, we present our experimental validation.

- **Chapter** 7 presents our approach that addresses reverse engineering SPLA from product variants. Firstly, we introduce a set of algorithms to identify mandatory features and the varia-

tion points of features as representative of commonality and variability at the feature level, respectively. Next, we extract components from the implementation of mandatory features and variation points of features. Then, we identify mandatory components and variation point of components by employing commonality and variability at the feature level. Finally, we present experimental results.

- **Chapter** 8 concludes the work presented in this thesis and highlight direct perspective work.
- **Appendices** IV include architecture implementation, components extraction tool used and product configuration used.

Part I

Background

CHAPTER 2

Preliminaries

Preamble

In this chapter, we present the main concepts in software product line engineering and techniques on which our work is built. We start by presenting software product line engineering, which represents our context. Next, we present information retrieval and formal concept analysis techniques, which we use in this thesis. We give the basic definitions for these techniques, supported with illustrative examples. Finally, we present case studies used throughout the thesis.

2.1 Software Product Line Engineering

Software Product Line Engineering (SPLE) emerges as an important development paradigm allowing companies to realize order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers [Clements et Northrop, 2001]. SPLE allows the development of multiple similar software systems from common software artifacts. In this section, we introduce the necessary background to understand SPLE

2.1.1 SPLE Definitions

SPLE is a software engineering discipline providing methods to promote systematic software reuse for developing short time-to-market and quality products in a cost-efficient way [Clements et Northrop, 2001]. These products are known as Software Product Line (SPL). The following definition proposed by [Clements et Northrop, 2001] captures the general idea behind a SPL:

"A software product line 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 that are developed from a common set of core assets in a prescribed way".

This definition relies on three main terms: software-intensive systems, core assets and features. Firstly, software-intensive systems refer to a family of similar software products, not a single software product. Secondly, core assets in a software engineering context, are a collection of reusable artifacts. These artifacts should be reused in a prescribed way to build applications (SPL members) for a particular context. Reusable artifacts include requirement models, architectural models, software components, test plans, etc. Finally, the concept of feature has many definitions in the literature, for example:

- [Kang *et al.*, 1990] :"a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems".
- [Kang *et al.*, 1998] :"a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained".
- [Czarnecki et Eisenecker, 2000] :"a distinguishable characteristic of a concept (e.g.,system, component, and so on) that is relevant to some stakeholder of the concept ".
- [Bosch, 2000] :"a logical unit of behavior specified by a set of functional and nonfunctional requirements ".
- [Batory *et al.*, 2003] :"a product characteristic that is used in distinguishing programs within a family of related programs ".
- [Zave et Jackson, 1997] :"an optional or incremental unit of functionality ".
- [Batory, 2005] :"an increment of program functionality ".

2.1.2 Benefits of SPLE

Compared to traditional single-product development, SPLs promise several benefits [Pohl *et al.*, 2010][Clements et Northrop, 2001]. In this section, we present key benefits that motivate software development under the SPLE:

- **Reduction of the Development Costs**
  The cost reduction is strongly correlated to SPLE which supports large-scale reuse during software development. In SPLE, software artifacts (core assets) are reused in several products; this implies a cost reduction for each product. Unfortunately, the reduction in the development cost does not come for free but it requires extra up-front investment [Linden *et al.*, 2007]. This is necessary in order to build SPL core assets which include all artifacts that can be reused in the future. After building SPL's core assets, the accumulated costs required to develop a large number of products are much less than developing them independently (traditional way for development) [Pohl *et al.*, 2010].

- **Quality Enhancement**
  The artifacts in the SPL core assets are reused and tested in many generated products. Therefore, they have proved their proper functioning in more than one product. The extensive quality assurance allows a significantly higher chance of detecting faults and correcting them, thereby increasing the quality of all SPL products [Pohl *et al.*, 2010].

- **Reduction of Time to Market**
  The time to market refers to the time to develop the product. Indeed, the time to market in SPLE is initially higher because the SPL's core assets have to be built first since the building process takes a long time. After building these core assets, the time to market is considerably reduced as many artifacts can be reused for each new product [Pohl *et al.*, 2010].

- **Reduction of Maintenance Efforts**
  Changing an artifact in the SPL's core assets, e.g. for the purpose of error correction, can be propagated to all products in which that artifact is being used. Therefore, there is no need to do the same task in each product like product variants. This may lead to reduced maintenance effort. Moreover, maintenance staff do not need to know all specific products and their parts and hence also reducing understanding effort [Pohl *et al.*, 2010].

2.1.3 Variability in SPLE

SPLE aims at providing a range of products. These products should support different and individual customer needs for a particular domain. As a result, variability is a key concept in SPLE. Variability, in general, means the tendency to change. *Weiss and Lai* define variability in SPLE as "an assumption about how members of a family may differ from each other " [Weiss et Lai, 1999]. Development of SPLs mainly relies on managing the variability between SPL's products to meet all customer requirements [Pohl *et al.*, 2010]. The main goal of variability is to "maximize the return on investment for building and maintaining products over a specified period of time or number of products" [Felix et Paul, 2005]. Variability can occur at all levels of abstraction, e.g., in requirement and architecture levels. Examples of variability include: the choice between two functional requirements and the

choice between multiple architectural styles. When we talk about variability in SPLE, the concept of commonality immediately stands out. Commonality refers to a characteristic (functional or non-functional) that can be common to all products in the product line [Linden *et al.*, 2007]. Usually, variability and commonality in SPLE are expressed in terms of features.

2.1.3.1 Variability Taxonomy

In the literature, there is no unique classification for variability. Pohl et al. [Pohl *et al.*, 2010] proposes variability *in time* and variability *in space*. They also propose *external* and *internal* variability. Halmans and Pohl distinguish *essential* and *technical* variability [Halmans et Pohl, 2003]. Below, we detail these classifications.

Variability in time is defined as "the existence of different versions of an artifact that are valid at different times" [Pohl *et al.*, 2010]. For example in the past, magnetic cards were used as door lock identification mechanisms, then fingerprints and then iris recognition. The variability in space is defined as "the existence of an artifact in different shapes at the same time " [Pohl *et al.*, 2010]. For example, an online banking system offers two options for communication with its customers: SMS or email.

External variability is defined as the variability that is visible to customers [Pohl *et al.*, 2010]. For example, customers of an online banking system can choose between two options for communication: SMS or email. Internal variability is defined as the variability that is hidden from customers [Pohl *et al.*, 2010]. For example, the choice between the major asymmetric encryption algorithms used for encrypting: RSA and DSA.

Essential variability describes the variability of a SPL in terms of customer point of view i.e., customer's wishes. For example, booking systems offer two options to issue a ticket: (i) before a travel takes places, e.g. before entering a train; (ii) during the travel, e.g. in the train. Technical variability deals with aspects about the realization of the variability. It corresponds to the SPL engineer viewpoint that cares about how to implement the variability. For example, such a booking system can be implemented with different types of database: ORACLE, MS access, etc.

2.1.3.2 Variability Modeling: Feature Model (FM)

The concept of feature is well-known in SPLE to describe variability because it is an agreement among all stakeholders about what a system should do [Passos *et al.*, 2013]. Also, feature is the first class entity in the SPLE to constitute core assets, as building SPL's core assets is driven by features that these assets should support. Therefore, feature model (FM) is commonly accepted language to model variability in SPLE. A FM is a representation of the requirements of a SPL abstracted at the feature level [Riebisch, 2003]. A feature can be a chunk of requirements that represents value to the end user. In SPLE, FM represents a hierarchy of features and includes constraints of feature selection. It models all possible products that can be generated in a given context so that each product is a valid combination of unique features. Figure 2.1 shows a simple FM inspired from the mobile phone industry [Benavides *et al.*, 2010]. Features in FM are represented as rectangles and can be optional or mandatory:

- An **optional feature** is a part of one or more products but not all products. It is denoted by an empty circle at the top (like *GPS* feature in Figure 2.1). Optional features represent *variability*.

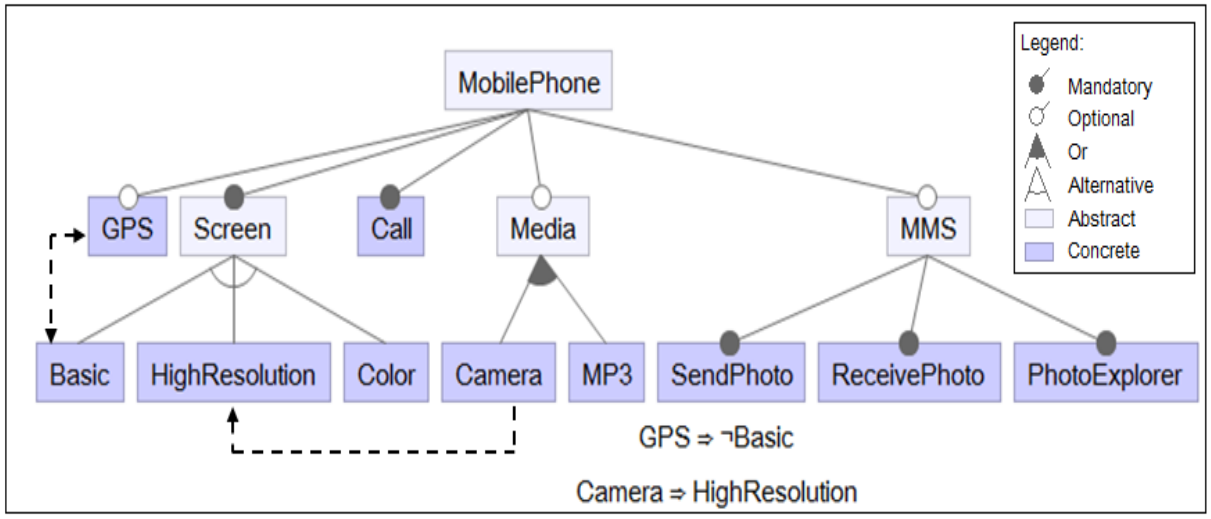

Figure 2.1 : FM of Mobile Phone SPL [Benavides *et al.*, 2010].

- A **mandatory feature** is a part of all products. It is denoted by a filled circle at the top (like *Call* feature in Figure 2.1). Mandatory features represent *commonality*.

An edge between features refers to a dependency. A solid edge is used for the feature tree edge while a dashed edge is used for a cross-tree constraint. Cross-tree constraints can be *exclude* or *require* [Benavides *et al.*, 2010]:

- **Exclude constraint**: if feature A excludes feature B, this means that both features cannot be part of the same product. For example, in Figure 2.1, *GPS* and *basic* are incompatible features, and hence they can not be members of the same product. This constraint can be represented graphically or textually (see Figure 2.1).
- **Require constraint**: if feature A requires feature B, this means that the inclusion of A in a product implies the inclusion of B in the same product. For example, in Figure 2.1, including a *camera* feature requires the inclusion of *high resolution screen* feature. This constraint can be represented graphically or textually (see Figure 2.1).

A feature may also be a member of a feature group. The parent of a group is called *abstract* feature while a member of a group is called a *concrete* feature. The abstract feature does not have an implementation because it is just a label. FM provides three types of feature groups [Benavides *et al.*, 2010]: *XOR-Group, OR-Group* and *AND-Group*.

- **XOR-Group**: a set of features has a XOR-relationship with their parent when only one of them can be selected in the case that their parent is selected. This type is marked with an empty arc in the FM. In Figure 2.1, the features *Basic*, *High Resolution* and *Color* represent an XOR-Group, as only one of them should be selected if their parent is selected.

- **OR-Group**: a set of features has an OR-relationship with their parent when one or more of them can be selected in the case that their parent is selected. This type is marked with a filled arc in the FM. In Figure 2.1, when *Media* is selected, *Camera*, *MP3* or both can be selected.

- **AND-Group**: a set of features has an AND-relationship with their parent when all of them must be selected in the case that their parent is selected. In Figure 2.1, the features *SendPhoto*, *ReceivePhoto* and *PhotoExplorer* represent an AND-Group, as all these features should be selected when their parent is selected.

Furthermore, optional features fallen down from the FM root represent a set of variants offered by the root. We consider these variants as a group of optional features and call them *OP-Group*. The FM's groups represent variation points (VPs) because each one offers a set of variants (features) to choose from.

FM models all possible products that can be generated in the future, so that each product is a valid combination of unique features. Each valid combination is called *product configuration*. Figure 2.2 shows an example of product configurations that can be obtained from FM shown in Figure 2.1.

```
Configuration 1 = { Call, MP3, Color, GPS }
Configuration 2 = { Call, Camera, HighResolution, GPS }
Configuration 3 = { Call, MP3, Color, Basic }
Configuration 4 = { Call, SendPhoto, ReceivePhoto, PhotoExplorer, MP3, Color }
Configuration 5 = { Call, MP3, Color, Basic }
 ...
```

Figure 2.2 : An Example of Product Configurations.

2.1.4 The SPLE Framework

Figure 2.3 shows a framework for SPLE. This framework is composed of two phases: *domain engineering* and *application engineering* [Pohl *et al.*, 2010] [Linden *et al.*, 2007]. In the following, we detail these phases:

2.1.4.1 Domain Engineering

Domain engineering refers to the creation process of the SPL's core assets. Therefore domain engineering is called a development *for reuse*. The domain engineering process (shown in the upper part of Figure 2.3) is composed of five key sub-processes: *product management*, *domain requirements engineering*, *domain design*, *domain realization* and *domain testing* [Pohl *et al.*, 2010].

Firstly, *product management* sub-process is concerned with the economic aspects of SPL and especially concerned with marketing strategy. It determines the SPL scope which represents a plan for the future development of the SPL's products. This plan determines the features of all SPL products and the schedule for marketing. Secondly, *domain requirement engineering* identifies and describes

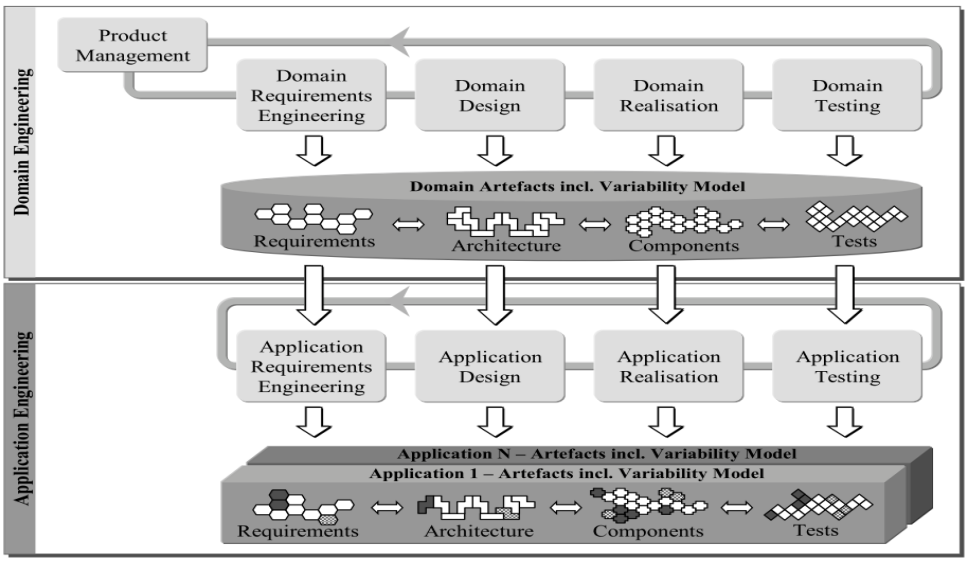

Figure 2.3 : SPL Framework [Pohl *et al.*, 2010].

the commonality and variability among potential SPL's products in terms of features. Next, *the domain design* sub-process includes all activities for building SPL architecture (SPLA). Next, *domain realization* is concerned with the implementation of reusable software components because SPL composed of component-based systems [Pohl *et al.*, 2010]. A component is "a unit of composition with contractually specified interfaces and explicit context dependencies only" [Szyperski, 2002]. A component is provided with two types of interfaces: *provided* and *required* interfaces. Provided interfaces determine what the component offers, whereas required interfaces determine what the component needs in order to do its job. Finally, *domain testing* performs validation and verification of reusable components. Domain testing checks the components against their specifications, i.e. requirements, architecture, etc. (see the return loop in upper part of Figure 2.3). These sub-processes produce different types of software artifacts (requirements, design models, components, test case, etc.). These artifacts together constitute SPL's core assets. According to [Pohl *et al.*, 2010], the key goals of the domain engineering process are to:

- Determine the set of products the SPL is planned for, i.e. define the scope of the SPL.
- Determine the commonality and the variability of SPL, i.e., determine features that are part of each product in SPL and features that are part from one product or more (but not all) in SPL.
- Build reusable software artifacts that accomplish the desired variability, i.e., build software artifacts that realize products for satisfying different customer needs.

2.1.4.2 Application Engineering

Application engineering refers to the process that uses those assets that are built in domain engineering to build SPL products. Hence, this process is called development *with reuse* [Linden *et al.*, 2007].

It is composed of four key sub-processes, in line with domain engineering sub-processes: *application requirements engineering*, *application design*, *application realization* and *application testing* (shown in the lower part of Figure 2.3) [Pohl *et al.*, 2010].

Firstly, *application requirements engineering* mainly specifies the requirements of a particular product. These requirements come from the stakeholders of the product under development, for example customers, product managers, etc. [Linden *et al.*, 2007]. Next, *application design* sub-process encompasses all activity needed to instantiate a product architecture from SPLA. Next, *application realization* sub-process implements the required product. It selects and assembles reusable components to create the required product. Finally, *application testing* sub-process encompasses the activities needed to validate and verify an application against its specification, i.e., requirements, architecture, etc. (see the return loop in upper part of Figure 2.3).

These sub-processes use as input software artifacts produced by the domain engineering process (requirements, architecture, components, test cases, etc.) to create applications (SPL's products) tailored to the specific needs of different customers. The application engineering process aims at exploiting commonality and variability in order to develop SPL members. It works by first determining the features that should be provided by some product (*product configuration*), then creates that product by using available SPL's core assets (*product derivation*) [Pohl *et al.*, 2010].

2.1.5 Different Approaches for SPL Development

Krueger [Krueger, 2002] propose three approaches for SPL development: *proactive*, *reactive* and *extractive* approaches.

In the proactive approach, organizations design and develop the core assets from scratch to support the full scope of products required for the foreseeable horizon. This is similar to the waterfall approach for single system development. In this approach, requirement analysis, design, source code, etc. are implemented, which leads to a risk of developing useless assets. Figure 2.4 shows the proactive approach.

In the reactive approach, organizations incrementally grow existing SPL's core assets when the demand arises for new products or new requirements for existing products. The common and varying software artifacts are incrementally extended in reaction to meet new demands of customers. This incremental approach provides a quicker and less expensive transition into SPL. Figure 2.5 shows the reactive approach.

In the extractive approach, organizations exploit the already existing product variants by extracting the common and varying software artifacts for building SPL core assets. This approach achieves short time-to-market products compared to other approaches. It also reduce significantly up-front investment, since a collection of assets are available and could be reused. Figure 2.6 shows the extractive approach.

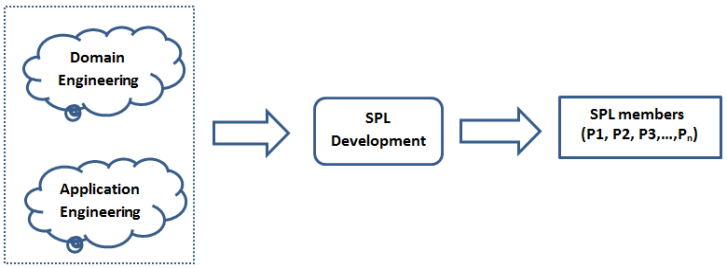

Figure 2.4 : Proactive Approach [de Almeida, 2010].

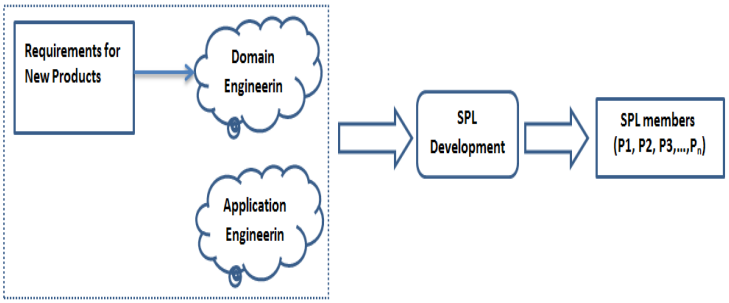

Figure 2.5 : Reactive Approach [de Almeida, 2010].

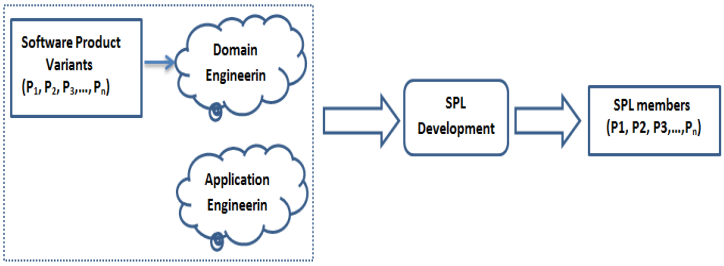

Figure 2.6 : Extractive Approach [de Almeida, 2010].

2.2 Information Retrieval Techniques

Information Retrieval (IR) provides many techniques to study the problem of identifying information or documents that are relevant to a query within a collection of documents [Grossman et Frieder, 2004]. Examples of such a collection in software engineering are source code files and requirement documents. Queries are formal statements of required information, for example strings describing source code bugs. IR works by finding a match (e.g., textual matching) between query information and documents information. It searches in a collection of documents using keywords extracted from

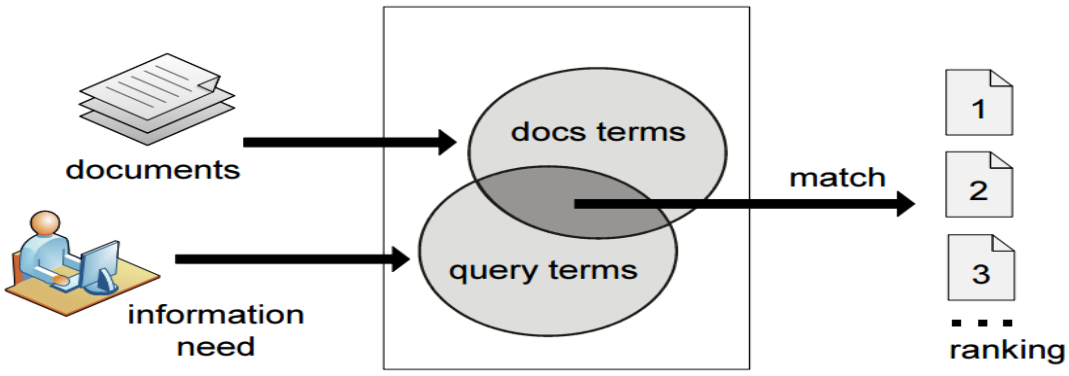

Figure 2.7 : Information Retrieval Process [Baeza-Yates et Ribeiro-Neto, 1999].

query information. In IR, a query does not uniquely match a single document in the collection. Instead, several documents may match the query with different degrees of relevance. Therefore, IR computes a numerical value to measure how each document in the collection matches the query and ranks the documents according to this value. The top ranking documents are then retrieved. Figure 2.7 shows IR process.

IR techniques have been used widely to support feature location [Bogdan *et al.*, 2013]. They are used to identify source code documents that are relevant to given features by lexical matching source code information with feature information. In this thesis, we use IR techniques to support feature location in a collection of software product variants. In the following, we present the main commonly used IR techniques for feature location, namely, Vector Space Model (VSM) and Latent Semantic Indexing (LSI). Also we present their evaluation metrics.

2.2.1 An Illustrative Example

As an illustrative example to clarify and exemplify IR techniques, we assume that there is a collection of three textual documents (D1, D2, D3) and a query (Q) as follows[1]:

- D1: “Shipment of gold damaged in a fire ”.
- D2: “Delivery of silver arrived in a silver truck ”.
- D3: “Shipment of gold arrived in a truck ”.
- Q: “Gold silver truck ”.

During our presentation of IR techniques, we attempt to compute similarity between the query (Q) and each document.

[1]This example is taken from [Grossman et Frieder, 2004]

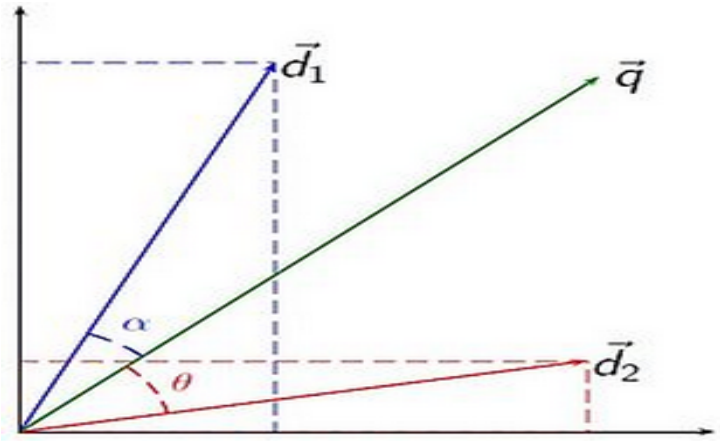

Figure 2.8 : An Example of Vector Representation in VSM.

2.2.2 Vector Space Model

In general, Vector Space Model (VSM) is an algebraic model for representing textual documents as vectors [Salton *et al.*, 1975]. In IR, VSM creates a space in which both documents and queries are represented by vectors in this space. The similarity between a query vector and a document vector is measured by the cosine of angle between them. Figure 2.8 shows an example of vector representation for two documents (d1, d2) and a query (q) in VSM.

The process of applying VSM consists of four steps [Grossman et Frieder, 2004]: *creating a corpus*, *preprocessing*, *indexing* and *computing similarity*. In the following, we detail these steps.

2.2.2.1 Corpus Creation

A corpus represents a collection of documents that contains the required information or documents. The type of these documents depends on the context that IR is applied to. For example, the in case of applying IR for feature location, the content of these documents represents source code information (identifiers and comments). In this case, a document can be created for each method, class, package or any other granularity of the source code. In this step, a document is created for each query containing information of that query, too.

2.2.2.2 Pre-processing

When the corpus and query documents are created, the text of each document must be pre-processed. The pre-processing step refers to the following options: (i) removing stop words (*or*, *the*, etc.); (ii) splitting compound words (*featuerLocation* becomes *feature* and *location*); (iii) performing stemming (e.g., *stemmed* becomes *stem)*. Also, query documents are pre-processed as corpus documents. The goal of pre-processing is to represent documents and queries effectively in terms of space (for saving the memory space of documents and queries). It also help to find better textual matching between queries and corpus documents by removing noise data (such as stop words), and hence improves the obtained results.

2.2.2.3 Indexing

In this step, the corpus is used to create *term-document* matrix with size $m \times n$ and *term-query* matrix with size $m \times l$ where m, n and l respectively represent number of all terms extracted from corpus query documents, number of documents in the corpus and number of query documents. Matrices' rows correspond to the terms while columns in *term-document* and *term-query* represent respectively corpus and query documents. A generic cell (i, j) in both matrices denotes the weight of the i^{th} term in the j^{th} document. Different measures were proposed for this weight [Salton et Buckley, 1988]. In the simple measures, the weight is a boolean value, either `1` if the i^{th} term occurs in the jth document, or `0` otherwise. In complex measures, the weight is computed based on the frequency of the terms in the documents, e.g., term frequency (TF). TF measures the number of a term occurrences in a document.

2.2.2.4 Computing Lexical Similarity

Once the corpus is indexed, each column in both *term-document* and *term-query* matrices represent a vector. Therefore, the similarity between a pair of corpus and query documents is typically measured by the cosine of the angle between their corresponding vectors. The cosine similarity takes a value in a range [`-1` to `1`]. The closer the cosine value is to one, the more similar the corpus document is to the query document. A cosine similarity value is computed between the query and each corpus document, and then the documents are descendant ranked by their similarity values. Equation 2.1 represents the cosine similarity equation [Grossman et Frieder, 2004]. Doc_q and Doc_c respectively refer to query and corpus documents while W_{qi} and W_{ci} refer respectively to the weight of the term i in the document (Doc_q) and the document (Doc_c).

$$COS(Doc_q, Doc_c) = \frac{\sum_{i=1}^{m} W_{qi} \times W_{ci}}{\sqrt{\sum_{i=1}^{m} (W_{qi})^2} \times \sqrt{\sum_{i=1}^{m} (W_{ci})^2}} \tag{2.1}$$

To apply VSM to our illustrative example, we assume that a term frequency is used as term weight for *term-document* and *term-query* matrices. Also, we assume the following preprocessing options: stop words were not ignored, text was tokenized and lowercased and no stemming was used.

Figure 2.9 shows *term-document* and *term-query* matrices of our illustrative example. By using equation 2.1 to compute cosine similarity values between the query (Q) and each document (D1, D2, D3), we obtain the following values for each document respectively: (`0.22`, `0.55`, `0.44`). From these values, we can notice that document D2 has a score higher than D3 and D1. This means that D2's vector is closer to the query vector than the other vectors. Therefore, we can rank these documents against the query (Q) based on their values in descending order as follows: D2 > D3 > D1. Actually, D2 has two shared terms (truck, silver) with the query (Q) and the "silver" are repeated two times in D2. D3 and D1 share with the query (Q) respectively two terms ("gold", "truck") and one term ("gold") without repetition.

2.2.3 Latent Semantic Indexing

Latent Semantic Indexing (LSI) is an advanced IR technique. It was developed as a solution for overcoming polysemy and synonymy problems that happen in VSM [Susan *et al.*, 1990]. These problems

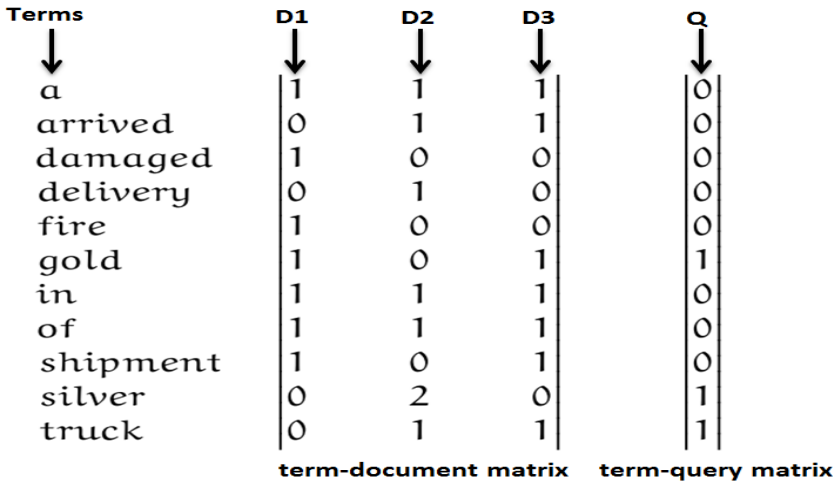

Figure 2.9 : Term-Document and Term-Query Matrices.

occur due to the fact that VSM does not take into account relations between terms. For instance, having a document containing a word "automobile" and another document containing a word "car" , does not contribute to finding similarity in these two documents. In LSI, the relations between terms, between documents and between terms and documents are explicitly taken into account [Lucia *et al.*, 2007]. LSI assumes that there is "latent structure" (co-occurrence of terms) in word usage pattern and it uses statistical analysis to find this structure. For example, both "car"and "automobile" probably co-occur in different documents with related terms, such as "motor", "wheel", etc. LSI uses information about co-occurrence of terms to find synonymy between different terms that cover the same concept.

LSI follows the VSM steps. Then, it applies Singular Value Decomposition (SVD) technique to *term-document* matrix. SVD aims to discover the latent structure between terms used in corpus and query documents [Susan *et al.*, 1990]. SVD is a mathematical technique originally used in signal processing to remove noise. SVD has a complex mathematical background, which hinders us to explain how SVD find the latent structure. The interested reader can refer to [Salton et McGill, 1986] for more details. In the following two steps, we present how SVD is applied to *term-document* matrix to perform his purpose.

2.2.3.1 Applying SVD Technique

SVD is applied to term-document matrix to decompose it into product of three other different matrices:

$$A = USV^T \tag{2.2}$$

where A is the *term-document* matrix with size $m \times n$, U is a matrix with size $m \times r$ containing the left singular vectors where r is the rank of A, S is a matrix with size $n \times n$ representing a diagonal matrix of singular values, V is a matrix with size $n \times r$ containing the right singular vectors. T refers to a transpose operation executed on V matrix. These matrices (U, S and V) are known as LSI sub-spaces. Figure 2.10 shows U, S and V^T matrices of *term-document* matrix shown in Figure 2.9.

$$U = \begin{vmatrix} 0.420 & -0.075 & -0.046 \\ 0.299 & 0.200 & 0.408 \\ 0.121 & -0.275 & -0.454 \\ 0.158 & 0.305 & -0.201 \\ 0.121 & -0.275 & -0.454 \\ 0.263 & -0.379 & 0.155 \\ 0.420 & -0.075 & -0.046 \\ 0.420 & -0.075 & -0.046 \\ 0.263 & -0.379 & 0.155 \\ 0.315 & 0.609 & -0.401 \\ 0.299 & 0.200 & 0.408 \end{vmatrix} S = \begin{vmatrix} 4.099 & 0.000 & 0.000 \\ 0.000 & 2.362 & 0.000 \\ 0.000 & 0.000 & 1.274 \end{vmatrix} V^T = \begin{vmatrix} 0.494 & 0.646 & 0.582 \\ -0.649 & 0.719 & -0.247 \\ -0.578 & -0.256 & 0.775 \end{vmatrix}$$

Figure 2.10 : The U, S, V^T Matrices of *Term-Document* Matrix Shown in Figure 2.9.

2.2.3.2 Reducing LSI Sub-Spaces

In this step, the U, S and V matrices are reduced in order to hold only the most significant relations between terms and documents, and at the same time removes the noise in words usage (e.g., stop words) that plagues LSI [Lucia *et al.*, 2007]. This reduction is performed by keeping only the first k columns of U and V, and the first k columns and rows of S where (k<r). The value of k is called the number of *term-topics*. A term-topic is a collection of terms that co-occur frequently in the documents of the corpus. The appropriate selection of k is critical. In fact, if the value of k is too much, this may lead to hold irrelevant relations. Also, if the value of k is too little, this may lead to loose relevant relations. The proper way to make such a choice is an open issue in the literature [Susan *et al.*, 1990][Dumais, 1992]. Figure 2.11 shows U, S and V matrices after the reduction where k=2.

$$U_k = \begin{vmatrix} 0.420 & -0.075 \\ 0.299 & 0.200 \\ 0.121 & -0.275 \\ 0.158 & 0.305 \\ 0.121 & -0.275 \\ 0.263 & -0.379 \\ 0.420 & -0.075 \\ 0.420 & -0.075 \\ 0.263 & -0.379 \\ 0.315 & 0.609 \\ 0.299 & 0.200 \end{vmatrix} S_k = \begin{vmatrix} 4.099 & 0.000 \\ 0.000 & 2.362 \end{vmatrix} V_k = \begin{vmatrix} 0.494 & -0.649 \\ 0.646 & 0.719 \\ 0.582 & -0.247 \end{vmatrix}$$

$$V_k^T = \begin{vmatrix} 0.494 & 0.646 & 0.582 \\ -0.649 & 0.719 & -0.247 \end{vmatrix}$$

Figure 2.11 : The U_k, S_k, V_k^T Matrices.

2.2.3.3 Computing Lexical Similarity

In LSI sub-spaces, new document and query vectors are created. For document vectors, each row in V_k matrix represents a document vector. By referring to V_k shown in Figure 2.11 D1, D2 and D3 have respectively the new following vectors: `(0.494, -0.649), (0.646, 0.719), (0.582, -0.247)`. For query vector, LSI uses the equation 2.3 to compute a new query vector. Based on this equation, the query (Q) shown in Figure 2.9 has the following new vector (`0.21, 0.18`).

$$Q = Q^T U_K S_K^{-1} \tag{2.3}$$

LSI computes lexical similarity between query vector and each document vector using cosine similarity like VSM (refers to equation 2.1). By using LSI, the similarity between the query (Q) and Documents D1, D2 and D3 respectively as follows: (`-0.054, 0.991, 0.448`). Based on this result, D2 is the closest to Q, then D3 and then D1.

2.2.4 Information Retrieval Performance Measures

As mentioned earlier, IR techniques return a ranked list of corpus documents which are similar to a given query in descending order. However, this list can include irrelevant documents (false-positive documents). Therefore, IR techniques provide two strategies to cut (i.e., to select documents) the ranked list in order to remove irrelevant documents as much as possible: (i) cutting the ranked list regardless of the similarity value [Lucia *et al.*, 2007]; (ii) cutting the ranked list using threshold on a similarity value. In the first strategy, there are two options as follows:

- **Constant cut point**: this option imposes a threshold on the number of documents that should be selected [Antoniol *et al.*, 2002], i.e., the top n documents of the ranked list are selected.
- **Variable cut point**: this option specifies a percentage of documents of the ranked list that should be selected.

In the second strategy, only the documents of ranked list having a similarity value greater than or equal to the specified threshold will be selected [Lucia *et al.*, 2007]. In this strategy there are three options as follows:

- **Constant threshold**: this option imposes a constant threshold regardless of the maximum similarity between documents and a given query. This is the most widely used in the literature [Lucia *et al.*, 2007]. The most commonly used threshold for cosine similarity is 0.70 [Marcus et Maletic, 2003a].
- **Variable threshold**: this option imposes a constant threshold in the interval [min similarity, max similarity] where *min* and *max* are the minimum and maximum similarity values in the ranked list [Lucia *et al.*, 2004].
- **Scale threshold**: in this option, the similarity threshold (Θ) should be computed as the percentage of the maximum similarity value that is obtained between documents and a given query, i.e., Θ = C·MaxSimilarity, where $0 \leq C \leq 1$ [Antoniol *et al.*, 2002].

The selected documents from the ranked list should be evaluated to measure the effectiveness of an IR technique. In the following subsections, we explain well-known IR metrics to evaluate the selected documents.

2.2.4.1 Precision and Recall

Precision and recall are well-known IR metrics to evaluate results obtained by IR techniques [Salton et McGill, 1986][Baeza-Yates et Ribeiro-Neto, 1999]. Both measures have values in the interval [0, 1]. Higher recall and precision values mean better results for the technique used. They are calculated for all documents retrieved above a threshold.

For a given query, precision is the percentage of relevant documents retrieved to the total number of retrieved documents. If the precision value is 1, this means that all the retrieved documents are relevant but also this does not mean that all relevant documents are retrieved (false negative documents). Equation 2.4 represents the precision metric equation.

$$Precision = \frac{|\{Relevant\ Documents\} \bigcap \{Retrieved\ Documents\}|}{|\{Retrieved\ Documents\}|} \times 100\% \tag{2.4}$$

For a given query, recall is the percentage of relevant documents retrieved to the total number of relevant documents. If the recall value is 1, this means that all relevant documents are retrieved. However, this does not mean that all retrieved documents are relevant (false positive documents). Equation 2.5 represents the recall metric equation.

$$Recall = \frac{|\{Relevant\ Documents\} \bigcap \{Retrieved\ Documents\}|}{|\{Relevant\ Documents\}|} \times 100\% \tag{2.5}$$

2.2.4.2 F-measure

F-measure is the harmonic mean of precision and recall that is computed as:

$$F\_measure = \frac{2}{\frac{1}{Reacll} + \frac{1}{Precision}} \times 100\% \tag{2.6}$$

The F-measure values take a range in [0, 1]. If F-measure value is 0, it means that no relevant documents have been retrieved. If F-measure value is 1, it means that all retrieved documents are relevant. Moreover, the harmonic mean (F-measure) gives a high value only when both recall and precision are high, too. Therefore, a high value of F-measure can be interpreted as an attempt to find the best possible compromise between recall and precision [Baeza-Yates et Ribeiro-Neto, 1999].

2.3 Formal Concept Analysis (FCA)

Formal Concept Analysis (FCA) is a technique for data analysis and knowledge representation based on lattice theory [Ganter et Wille, 1999]. It identifies meaningful groups of objects sharing common attributes and provides a theoretical model to analyze hierarchies of these groups. This technique is currently applied to support various tasks, including locating feature implementations [Poshyvanyk et Marcus, 2007], data mining [Valtchev *et al.*, 2004], building or maintaining class hierarchies in object-oriented software [Godin et Mili, 1998], software understanding [Bhatti *et al.*, 2012]. In this section, we present the definitions of FCA.

2.3.1 An Illustrative Example

We explain definitions of the FCA technique along with an illustrative example. Suppose we have a list of Mexican dishes and a list of ingredients for each dish. We assume the data set in Table 2.1 for these Mexican dishes. Rows represents dishes while the column represents the list of ingredients of each dish.

Table 2.1 : Mexican dishes and their ingredients.

Mexican dish	**Ingredients**
Burritos	chicken, beef, pork, vegetables, beans, rice, cheese, guacamole, sour-cream, lettuce and flour-tortilla
Enchiladas	chicken, cheese, sour-cream and corn-tortilla
Fajitas	chicken, beef, vegetables, cheese, guacamole, sour-cream, lettuce and flour-tortilla
Nachos	vegetables, beans, cheese and guacamole
Quesadillas	chicken, beef, cheese, corn-tortilla and flour-tortilla
Tacos	chicken, beef, beans, cheese, lettuce, corn-tortilla and flour-tortilla

2.3.2 Definitions

Definition 1 (Formal Context) *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, i.e.,* $R \subseteq O \times A$.

A formal context is represented as a cross table in which row labels display objects and column labels display attributes. Table 2.2 shows a formal context for Mexican dishes and their ingredients. A cross in the cell *(o, a)* of this table refers to the object *o* that possesses the attribute *a*. The key modeling issue that strongly impacts on the analysis performed by FCA is the right choosing of objects, attributes and relations. From our illustrative example, we can create a formal context of Mexican dishes O={*Burritos, Enchiladas, Fajitas, Nachos, Quesadillas, Tacos*} and their ingredients A={*chicken, beef, pork, vegetables, beans, rice, cheese, guacamole, sour-cream, lettuce, corn-tortilla, flour-tortilla*}.

For a set of objects $M \subseteq O$, then $M' = \{a \in A | \forall o \in M : (o, a) \in R\}$ is the set of common attributes. Also, for a set of attributes $B \subseteq A$, then $B' = \{o \in O | \forall a \in B : (o, a) \in R\}$ is the set of common objects. For example, if we take the set M = {*Enchiladas, Quesadillas, Tacos*} from Table 2.2, the set of com-

Table 2.2 : A formal context for Mexican dishes.

	chicken	beef	pork	vegetables	beans	rice	cheese	guacamole	sour-cream	lettuce	corn-tortilla	flour-tortilla
Burritos	X	X	X	X	X	X	X	X	X	X		X
Enchiladas	X						X		X		X	
Fajitas	X	X		X			X	X	X	X		X
Nachos				X	X		X	X				
Quesadillas	X	X					X				X	X
Tacos	X	X			X		X			X	X	X

mon attributes is M' ={*chicken, cheese, corn tortilla*}. In the same way, if B = {*pork, rice*} then, B' = {*Burritos*}.

Definition 2 (Formal Concept) *For a given formal context $K = (O, A, R)$, a formal concept is a pair (E, I) composed of an object set $E \subseteq O$ and an attribute set $I \subseteq A$. $E = \{o \in O | \forall a \in I, (o, a) \in R\}$ is the extent of the concept (i.e., the objects covered by the concept). $I = \{a \in A | \forall o \in E, (o, a) \in R\}$ is the intent of the concept (i.e., the attributes shared by the objects covered by the concept).*

For example, (*{Enchiladas, Quesadillas, Tacos}, {chicken, cheese, corn-tortilla}*) is a concept, while (*{ Nachos } , {vegetables, beans, cheese, guacamole}*) is not, because (*{ Nachos}*)' = *{vegetables, beans, cheese, guacamole}* while (*{vegetables, beans, cheese, guacamole}*)' = *{Burritos, Nachos}*.

Definition 3 (Concept Specialization Order) *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.*

For example, ({Burritos}, {chicken, beef, pork, vegetables, beans, rice, cheese, guacamole, sour-cream, lettuce, flour-tortilla}) is a sub-concept of ({Burritos, Nachos}, {vegetables, beans, cheese, guacamole}). Due to this specialization order definition, an evident property is that a sub-concept owns (inherits in top-down manner) the attributes of its super-concepts, while a super-concept owns (inherits in bottom-up manner) the objects of its sub-concepts.

Definition 4 (Concept Lattice) *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 2.12 shows a part of lattice concept associated with the formal context of Table 2.2. This sub-order of the lattice is known the Galois Sub-Hierarchy (GSH, also called AOC-Poset) [Berry *et al.*,

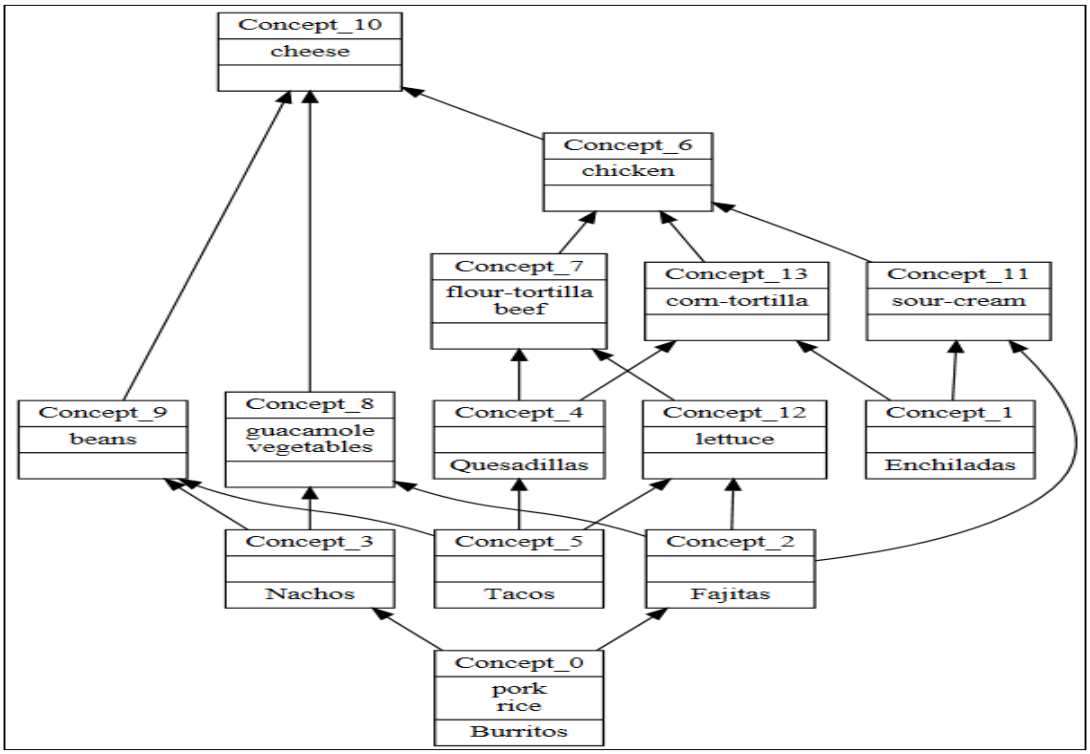

Figure 2.12 : GSH for the Context in Table 2.2.

2014]. In this GSH we can discover many facts as well as the relationships between the presented Mexican dishes, for example:

- All the Mexican dishes contain *cheese* because it appears as the intent of the top concept (*Concept_10*) that has all the dishes in its extent.
- When a concept has only Mexican dishes in its extent, this means that these dishes don't have specific ingredients and they share ingredients with other dishes belonging to different concepts, for example: in *Concept_3*, *Nachos* inherits *vegetables*, *guacamole*, *beans* and *cheese*.
- When a concept has only ingredients in its intent, this means that these ingredients are not specific for a certain Mexican dish but they are shared among different dishes which belong to multiple concepts, for example: in *Concept_9*, *beans* is shared between *Nachos*, *Tacos* and *Burritos* dishes.

Galois lattices (or concept lattices) are a powerful tool for knowledge representation. Most of the algorithms for building concept lattices are cited and compared in [Kuznetsov et Obiedkov, 2002]. The main drawback of this structure is that it may have an exponential size in the number of objects or attributes [Berry *et al.*, 2014]. In this thesis, we rely on GSH as the best running time of one of algorithms proposed to build GSH is O(min{nm, $n^{2.376}$ } where n is the number of objects or attributes

while m is the size of the relation [Berry *et al.*, 2014]. This major difference in terms of running time between concept lattice and GSH due to the fact that GSH has non empty concepts (empty extent and intent). This leads to a drastic difference of complexity between the two structures, because the concept lattice may have $2^{min(|O|,|A|)}$ concepts, while the number of concepts in the GSH is bounded by |O| + |A| [Berry *et al.*, 2014].

2.4 Case Studies

We use through this thesis three case studies: ArgoUML-SPL[2], a large-scale system, BerekelyDB-SPL[3], a medium-scale system, and MobileMedia[4], a small-scale system.

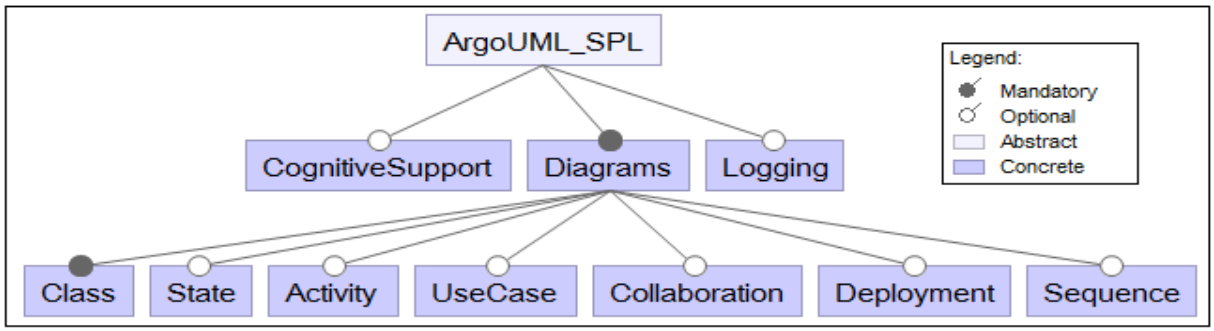

Figure 2.13 : FM of ArgoUML-SPL.

2.4.1 ArgoUML-SPL

ArgoUML-SPL is well-known case study in our context [Xue *et al.*, 2012][Ziadi *et al.*, 2012][Couto *et al.*, 2011]. It is a JAVA open-source which supports all standard UML diagrams. Figure 2.13 shows the FM of ArgoUML-SPL. This FM shows that *Class Diagram* is mandatory feature. It also shows that most of the features are organized as a single OR-Group (*Class, State, Activity, UseCase, Collaboration, Deployment* and *Sequence* diagrams) while others represent members of an OP-Group (*Cognitive Support* and *Logging*). The implementation of *Cognitive Support* and *Logging* have crosscutting behavior through all other feature implementations [Couto *et al.*, 2011]. *Cognitive Support* feature detects design errors made by end users while *logging* supports debug logging and tracing messages raised by UML diagrams.

We have used seven products of ArgoUML-SPL. These products cover all features shown in Figure 2.13 and are generated from the same codebase so that products that share some features also share the same code. These products also have two common features (*Class Diagram* and *Cognitive Support*) and differ in other features. Table 2.3 shows configurations of ArgoUML-SPL products. Table 2.4 shows statistical information of source code of ArgoUML-SPL's products in terms of number

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

Table 2.3 : Product configuration of ArgoUML-SPL's products.

Products	Configuration
Product1	Class, Cognitive, Sequence, Collaboration, Deployment, Usecase, Activity, State.
Product2	Class, Cognitive, Sequence, Collaboration, Deployment, Usecase.
Product3	Class, Cognitive, Sequence, Activity, State, Usecase.
Product4	Class, Cognitive, Activity, Collaboration, Deployment, State.
Product5	Class, Cognitive.
Product6	Class, Cognitive, Sequence, Usecase.
Product7	Class, Cognitive, Collaboration, Deployment.

Table 2.4 : Statical information of source code of ArgoUML-SPL's products.

Products	NOP	NOC	LOC
Product1	81	1666	118189
Product2	74	1587	112060
Product3	72	1607	113533
Product4	69	1541	110168
Product5	63	1455	99243
Product6	70	1554	107334
Product7	67	1488	103969

of packages (*NOP*), number of classes (*NOC*) and number of lines of code (*LOC*). The *logging* feature is excluded from this table because it is implemented by an external library called Log4J [Couto *et al.*, 2011]. ArgoUML-SPL's features are mainly implemented by source code classes. To establish ground truth links between features and their source code classes, we perform the following process. Considering a given feature (*F*), we generate two products of ArgoUML-SPL such that one of them (*P1*) provides all features and other (*P2*) provides all features except the focused one (*F*). Then, we compare code classes of two generated products. The classes that are present in *P1* and absent in *P2* represent the real implementation of *F*. We repeat this process to obtain real implementation for all ArgoUML-SPL's features. For feature descriptions, we obtain them based on ArgoUML-SPL official website and manual instructions.

2.4.2 MobileMedia

The MobileMedia is a JAVA open source software which manipulates multimedia on mobile devices. It was implemented in *8* subsequent releases, each incorporates new mandatory, optional or alternative features. Each release represents a variant corresponds to an evolutionary step of the system development. Table 2.5 summaries changes made to each release. We only consider releases (1-3 and 5-6) and exclude (R4, R7, R8) due to the nature of the evolution. R4 is produced by evolving the R3 through adding *Set Favorite, View Favorite* and *Sorting* to R3. These features are implemented by modifying already existing source code classes, which means they are implemented using lower

Table 2.5 : Summary of evolution in MobileMedia.

Release	Description	Type of Change
R1	MobilePhoto core (album and photo management)	
R2	Exception handling included	Inclusion of non-functional concern.
R3	New feature added to count the number of times a photo has been viewed and sorting photos by highest viewing frequency. New feature added to edit the photo's label.	Inclusion of optional and mandatory features.
R4	New feature added to allow users to specify and view their favorite photos.	Inclusion of optional feature.
R5	New feature added to allow users to keep multiple copies of photos.	Inclusion of optional feature.
R6	New feature added to send photo to other users by SMS.	Inclusion of optional feature.
R7	New feature added to store, play, and organize music. The management of photo (e.g. create, delete and label) was turned into an alternative feature. All extended functionalities (e.g. sorting, favourites and SMS transfer) were also provided	Changing of one mandatory feature into two alternatives.
R8	New feature added to manage videos	Inclusion of alternative feature

source code granularity (such as methods) but not classes. We ignore the R4 because we assume that in this thesis a feature is implemented by a set of classes. The evolution in *R7* and *R8* involve merging two or more features (resp. their implementations) together. This means that the same feature does not has the same implementation in *R7* and *R8*, and hence violate our assumption about the feature implementations (see section 5.2). Figure 2.14 shows FM of the releases considered.

Table 2.6 shows variant configurations of the MobileMedia. In these variants, there are group of features that are implemented by the same set of classes with differences in method bodies. These groups are: (*Create Album* and *Delete Album*), (*Create Photo, Delete Photo, View Photo*) and (*Send Photo* and *Receive Photo*). We consider the parent of these group as representative of these features. For example as shown in MobileMedia's FM, *AlbumManagment* is a representative of (*Create Album* and *Delete Album*). This means that we locate the implementation of the representative features. To establish ground truth links between features and their source code classes, we analyze manually the source code. The description of MobileMedia[5] features are obtained by official website of MobileMedia, descriptions of its use cases and analyzing source code comments. Table 2.7 shows statistical information for MobileMedia source code in terms of number of packages (NOP), number of classes (NOC) and number of line of code (LOC).

2.4.3 BerkeleyDB-SPL

BerkeleyDB-SPL is an open source database engine, entirely implemented in JAVA. It can work as a standalone database (run as .jar file), or be embedded as a third party library in the JAVA application. It provides the embedded storage, with open interfaces designed for programmers. Originally, BerkeleyDB was a single application, but *Käst*ner and others have re-engineered it as a SPL [Christian, 2007]. Core assets of BerkeleyDB-SPL implement *(42)* features. Most of these features are implemented by low source code granularity (such as methods) [Christian, 2007]. There are only *(25)*

[5]http://www.ic.unicamp.br/ tizzei/mobilemedia/

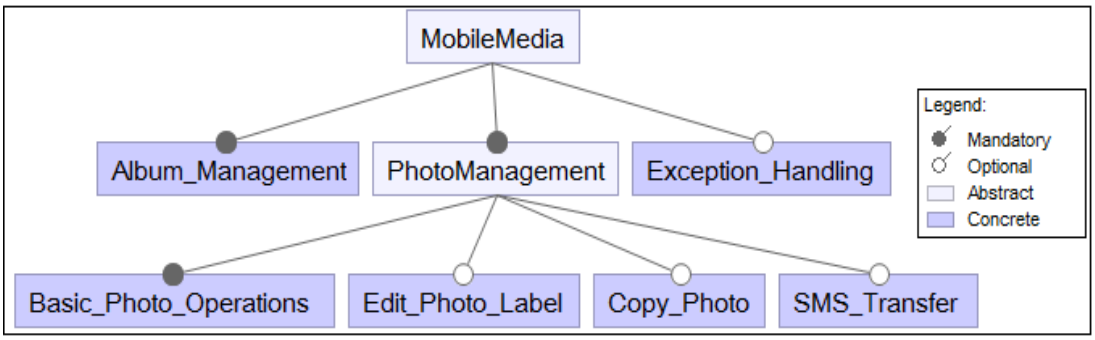

Figure 2.14 : FM of Considered MobileMedia Releases.

Table 2.6 : Product configuration for MobileMedi's releases.

Release	Configuration
R1	Album Management (Create Album, Delete Album), Basic Photo Operations (Create Photo, Delete Photo, View Photo).
R2	Exception Handling, Album Management, Basic Photo Operations.
R3	Exception Handling, Edit Photo Label, Album Management , Basic Photo Operations.
R5	Exception Handling, Copy Photo, Edit Photo Label, Album Management, Basic Photo Operations.
R6	Exception Handling, SMS Transfer (Send Photo, Receive Photo), Copy Photo, Edit Photo Label, Album Management, Photo Management.

Table 2.7 : Statical information of source code of MobileMedia's releases.

Products	NOP	NOC	LOC
Release1	6	15	1,758
Release2	7	24	2,182
Release3	7	25	2,481
Release5	7	31	2,969
Release6	9	38	3,889

features that their associated classes are known and available[6]. Therefore, we create core assets only containing these features. Table 2.8 presents the features considered. Figure 2.15 shows FM of BerkeleyDB-SPL.

[6]http://www.fosd.de/FeatureVisu/

Table 2.8 : Features of BerkeleyDB-SPL

Features		
CheckLeaks	EnvironmentLocking	NIO
CheckpointerDaemon	Evictor	RenameOp
Checksum	FileHandleCache	Statistics
ChunkedNIO	FSync	TruncateOp
CleanerDaemon	INCompressor	Verifier
CPBytes	IO	
CPTime	Latches	
CriticalEviction	Logging	
DeleteOp	LookAHEADCache	
DiskFullErro	MemoryBudget	

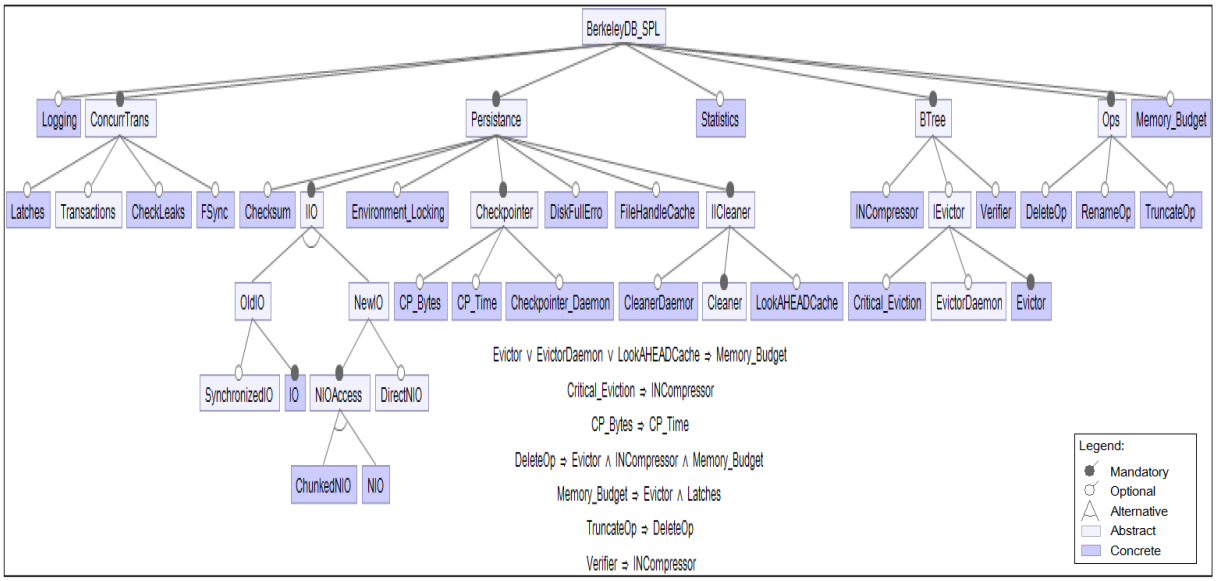

Figure 2.15 : FM of BerkeleyDB-SPL.

2.5 Summary

In this chapter, we presented fundamental concepts in software product line engineering, techniques on which our work is built on and case studies used in the evaluation. We first described the software product line engineering framework and detailed main processes in this framework: *domain* and *application engineering* processes. We explained the notion of variability and its importance in SPLE. Next, we presented necessary background about IR techniques, namely LSI and VSM. We described the steps of these techniques and their evaluation metrics. Then, we explained the basic definitions in FCA along with an illustrative example. Finally, we described case studies used in the experimental evaluation during the thesis.

Part II

State-Of-The-Art

CHAPTER 3

IR-Based Feature Location

Preamble

In this chapter, we present a state-of-the-art IR-based feature location. In section 3.1, we present types of software traceability links to understand the type of links identified by our approach. In section 3.2, we give a general classification of feature location approaches. Section 3.3 focuses on IR-based feature location, as our approach relies on IR to locate feature implementations in a collection of product variants. In section 3.4, we present a comparison of IR-based feature location approaches, following a set of criteria that are relevant to our contribution. Section 3.5 concludes this chapter by showing the need to propose a new IR-based feature location in a collection of product variants.

3.1 Traceability Links Types

Software traceability "is the ability to relate artifacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artifacts, and the rationale that explains the form of the artifacts" [George et Andrea, 2004]. Software traceability refers to different types of traceability links. These different types have been classified based on two criteria [Robert et Matthias, 2008][George et Andrea, 2004]: (i) the semantics associated with each type of link, and (ii) the abstraction level of the linked software artifacts.

3.1.1 The Semantics Associated with each Type of Link

Based on this criterion, there are nine types of traceability links: *dependency, generalization/refinement, evolution, satisfaction, overlapping, conflicting, rationalization, contribution and realization (implementation)* [Robert et Matthias, 2008] [George et Andrea, 2004].

- **Dependency link**: In this type, a software artifact (*a1*) depends on another software artifact (*a2*), if the existence of *a1* relies on the existence of *a2* or if changes made to *a2* have to be propagated to *a1*. Dependencies between use cases in UseCase diagrams is an example of this type.
- **Generalization/refinement link**: It refers to how a compound software artifact of a system can be split into sub-artifacts, how sub-artifacts of a system can be combined to form a compound artifact and how an artifact can refine another one. Dividing a requirement into sub-requirements is an example of such a type.
- **Evolution link**: It refers to the evolution of software artifacts. In this type, a software artifact (*a1*) evolves into a software artifact (*a2*), if *a1* is replaced by *a2* during the maintenance or evolution of the system. Replacing a source code class with another class during the maintenance is an example of this type.
- **Satisfaction link**: In this type, a software artifact (*a1*) satisfies a software artifact (*a2*), if *a1* meets the expectation, needs and desires of *a2*; or if *a1* complies with a condition represented by *a2*. Linking requirements to the system components (e.g. architectural components) that satisfy these requirements is an example of this type.
- **Overlap link**: In this type, a software artifact (*a1*) overlaps with another software artifact (*a2*), if *a1* and *a2* have common parts or characteristics. Textual matching between code information (comments and identifiers) and textual description of requirement is an example of this type.
- **Conflict link**: It refers to conflicts between two software artifacts. For example, when two requirements (functionalities) conflict with each other.
- **Rationalization link**: It refers to the rationale behind the creation and evolution of software artifacts through different levels of abstraction. For example, adding a new requirement to the existing requirements involves changing the implementation of other requirements.

- **Contribution link**: It refers to the associations between requirements and stakeholders who have contributed to generate the requirements. The relation between customers' wishes and system requirements is an example of this type because customers' wishes represent a trigger for the creation of system requirements.

- **Realization or implementation link**: It refers to implementation link between software artifacts across different levels of abstraction. Linking a system requirement (*r1*) to a portion of source code that implement (*r1*) is an example of this type.

In this classification, it is possible that the same link has more than one meaning (semantic). For example, the textual matching between source code information and textual information of requirements refers to *overlap* and *implementation* links.

3.1.2 The Level of Abstraction of the Linked Software Artifacts

Depending on whether traceability links relate software artifacts of the same level of abstraction or different levels, traceability links can be classified into *horizontal* or *vertical* links [Lindvall et Sandahl, 1996][Boldyreff *et al.*, 1996]. The former refers to links among software artifacts belonging to the same level of abstraction (e.g., among related requirements). The later refers to links between software artifacts belonging to different levels of abstractions (e.g., between requirements and source code elements). Sometimes, the authors swap the definition of horizontal and vertical ([Lindvall et Sandahl, 1996], e.g. compare [Boldyreff *et al.*, 1996]). Figure 3.1 shows an example of horizontal and vertical traceability links.

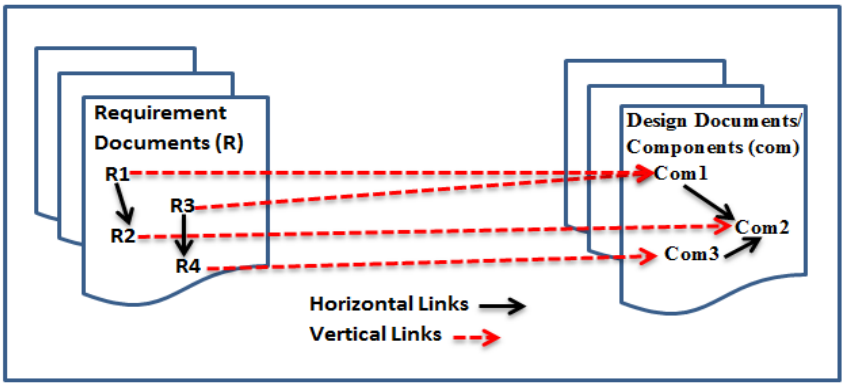

Figure 3.1 : An Example of Vertical and Horizontal Traceability Links.

3.1.3 Variability Traceability in SPLE

In addition to horizontal and vertical traceability links, variability in SPLE adds a third dimension to these links by linking elements (features) of FM as representative of the variability in SPLE to elements of other models at different levels of abstraction [Anquetil *et al.*, 2010][Pohl *et al.*, 2010]. This dimension is called *variability traceability* [Pohl *et al.*, 2010]. There are two types of variability traceability:

(i) variability traceability among only the domain engineering artifacts, (ii) variability traceability between domain engineering and application engineering artifacts. The former is used "to relate the variability defined in the variability model (FM) to software artifacts specified in other models, textual documents, and code" [Pohl *et al.*, 2010]. Such traceability links are restricted to SPL core asset artifacts (domain engineering) where variability is defined. Solid circles in Figure 3.2 refer to this type. The later is used "to link application artifacts to the underlying domain artifact" [Pohl *et al.*, 2010]. Such type of variability traceability is used to relate application engineering artifacts (artifacts for a specific product, application, in a SPL) to domain engineering artifacts. Dashed circles in Figure 3.2 refer to the second type of variability traceability.

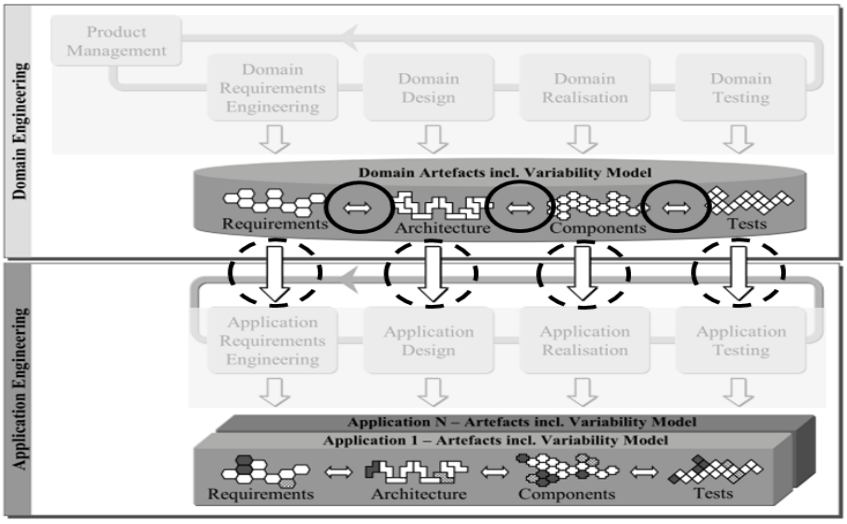

Figure 3.2 : Software Product Line Framework [Pohl *et al.*, 2010].

3.2 Classification of Feature Location Approaches

As mentioned earlier, the well-known term of the process of finding traceability links between features and source code elements that implement those features is *feature location*. Feature location is one of the most important and common activities performed by programmers during software maintenance and evolution, as any maintenance activity can not be performed with first understanding and locating the code relevant to the task at hand [Bogdan *et al.*, 2013]. There is a large body of research on feature location approaches [Bogdan *et al.*, 2013][Cornelissen *et al.*, 2009]. The distinguishing factor between these approaches is the type of analysis that they use. The various types of analysis refer to *dynamic*, *static* and *textual*.

3.2.1 Dynamic-Based Feature Location Approaches

Dynamic analysis refers to collecting information from a system during runtime. For the purpose of feature location, it is used to locate feature implementations that can be called during runtime by test scenarios [Koschke et Quante, 2005][Asadi *et al.*, 2010]. Feature location using dynamic analysis

depends on the analysis of execution traces. An execution trace is a sequence of source code entities (classes, methods, etc.). Usually, one or more feature-specific scenarios are developed that invoke only the implementation of feature of interest. Then, the scenarios are run and execution traces are collected, recording information about the code that was invoked. These traces are obtained by instrumenting the system code. Using dynamic analysis, the source code elements pertaining to a feature can be determined in several ways. Comparing the traces of the feature of interest to other feature traces in order to find source code elements that only is invoked in the feature-specific traces [Eisenbarth *et al.*, 2003][Wilde et Scully, 1995]. Alternatively, the frequency of execution parts of source code can be analyzed to determine the implementation of a feature [Eisenberg et De Volder, 2005][Safyallah et Sartipi, 2006]. For example, a method exercised multiple times and in different situations by test scenarios relevant to a feature is more likely to be relevant to the feature being located than a method used less often.

Feature location by using dynamic analysis has some limitations. The test scenarios used to collect traces may invoke some but not all the code portions that are relevant to a given feature; this means that some of the implementation of that feature may not be located. Moreover, it may be difficult to formulate a scenario that invokes only the required feature, which leads to obtain irrelevant source code elements. Additionally, developing test scenarios involves well-documented systems to understand the system functionalities [Bogdan *et al.*, 2013]. Such maintainers may be not always available, especially in legacy product variants.

3.2.2 Static-Based Feature Location Approaches

Feature location using static analysis refers to the analysis of the source code to explore structural information such as control or data flow dependencies. Static feature location approaches require not only dependence graphs, but also a set of source code elements which serve as a starting point for the analysis. This initial set is relevant to features of interest and usually specified by maintainers. The role of static analysis is to determine other source code elements relevant to the initial set using dependency graphs [Kunrong et Václav, 2000].

Static approaches allow maintainers to be very close to what they are searching for in the source code, as they start from source code elements (initial set) specific to a feature of interest. However, these approaches often exceed what is pertinent to a feature and are prone to returning irrelevant code [Bogdan *et al.*, 2013]. This is because following all dependencies of a section of code that is relevant to a feature may catch source code elements that are irrelevant. In addition, static approaches need maintainers who are familiar with the code in order to determine the initial set.

3.2.3 Textual-Based Feature Location Approaches

Textual information embedded in source code comments and identifiers provides important guidance about where features are implemented. Feature location using textual analysis aims to analyze this information to locate a feature's implementation [Savage *et al.*, 2010]. This analysis is performed by three different ways: *pattern matching, Natural Language Processing (NLP)* and *Information Retrieval (IR)*.

3.2.3.1 Pattern Matching

Pattern matching usually needs a textual search inside a given source code using a utility tool, such as *grep* [Petrenko *et al.*, 2008]. Maintainers formulate a query that describes a feature to be located then they use a pattern matching tool to investigate lines of code that match the query. Pattern matching is not very precise due to the vocabulary problem; the probability of choosing a query's terms, using unfamiliar source code maintainers, that match the source code vocabulary is relatively low [Furnas *et al.*, 1987].

3.2.3.2 Natural Language Processing

NLP-based feature location approaches analyze the parts of the words (such as noun phrases, verb phrases and prepositional phrases) used in the source code [Shepherd *et al.*, 2006]. They rely on the assumption that verbs in object-oriented programs correspond to methods, whereas nouns correspond to objects. As an input for these approaches, the user formulates a query describing the feature of interest and then the content of the query is decomposed into a set of pairs *(verb, object)*. These approaches work by finding methods and objects inside the source code, which are similar to the input verbs and objects, respectively. NLP is more precise than pattern matching but relatively expensive [Bogdan *et al.*, 2013].

3.2.3.3 Information Retrieval

As mentioned in preliminaries chapter, IR-based techniques, such as LSI and VSM, are textual matching techniques to find textual similarity between a query and given corpus of textual documents. For the purpose of locating a feature's implementation, a feature's description represents the subject of a query while source code documents represent corpus documents. A feature description is a natural language description consisting of short paragraph(s). A source code document contains textual information of certain granularity of source code, such as a method, a class or a package. IR-based feature location approaches find a code portion that is relevant to the feature of interest by conducting a textual matching between identifiers and comments of a given source code portion and the description of the feature to be located [Lucia *et al.*, 2007]. IR lies between NLP and pattern matching in terms of accuracy and complexity [Bogdan *et al.*, 2013].

Regardless of the type of textual analysis used (pattern matching, NLP and IR), generally the quality of these approaches mainly depends on the quality of the source code naming conventions and the query. In the remainder of this chapter, we focus only on IR-based feature location approaches because our feature location approach aims at improving the performance of IR for locating feature implementations in a collection of product variants.

3.3 IR-based Feature Location Approaches

The application of IR techniques in software engineering has been considered by several researchers [Poshyvanyk *et al.*, 2007][Xue *et al.*, 2012][Poshyvanyk et Marcus, 2007][Marcus *et al.*, 2004]. One of the most important applications of IR in software engineering is the feature location [Lucia *et al.*, 2007]. We classify the IR-based feature location approaches into two categories based on how they

deal with product variants. The first category includes all approaches that consider product variants as singular independent products. We call this group *feature location in single software product*. The second category includes all approaches considering product variants as a collection of similar and related products. We call this group *feature location in a collection of product variants*.

3.3.1 Feature Location in Single Software Product with IR

All approaches that belong to this category represent the conventional application of IR for the purpose of locating feature implementations. Figure 3.3 shows an example of the conventional application of IR in a collection of three product variants. We notice that this application involves conducting a textual matching between all features (i.e., their descriptions) and entire source code of each product in product variants independently. These features and the source code represent the *IR search spaces*. In case of having a large-scale system, such an application of IR may lead to retrieve irrelevant source code elements for features, especially in case of having similar features and trivial descriptions. In the literature, different strategies have been proposed to improve the performance of this conventional application of IR concerned with locating feature implementations. In the beginning, VSM was used for this purpose. Then, LSI was proposed to overcome polysemy and synonymy problems that occur in VSM. Then, the performance of both LSI and VSM is improved using other source code information such as static information, dynamic information or their combinations. Below, we list the approaches of this category based on the strategy used in chronological order.

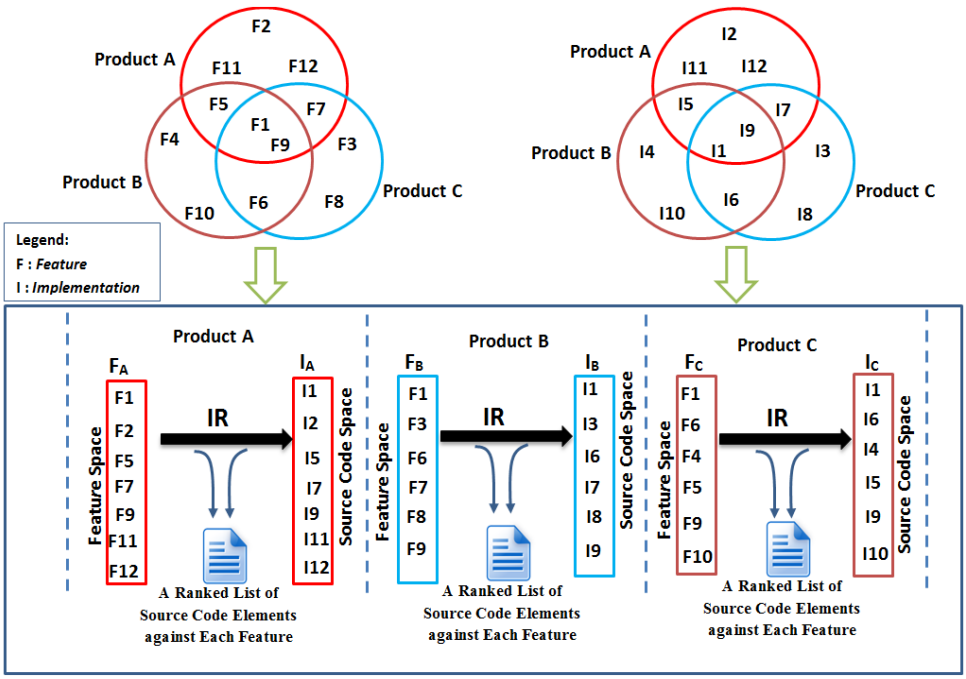

Figure 3.3 : An Example of the Conventional Application of IR for Locating Feature Implementations in a Collection of Three Product Variants.

3.3.1.1 VSM-based Feature Location

In [Antoniol *et al.*, 2002], *Antoniol et al.* who first used VSM to recover traceability links between source code and free text documentation (e.g., requirement specifications, design documents, error logs, etc.). In their approach, VSM is applied to trace C++ and Java source classes to manual pages and functional requirements, respectively. In their approach, documentation pages were used to create IR's corpus documents while source code files are used as queries. They compare the identifiers in source code and the words in corpus documents to recover traceability links.

In [Gay *et al.*, 2009], *Gay et al.* introduce the notion of relevance feedback from maintainers into feature location process to improve VSM results. Relevance feedback represent a maintainer feedback about the list of source code elements retrieved by VSM. After VSM returns a ranked list of source code elements relevant to a query (e.g., feature description), the maintainer ranks the top n source code elements as relevant or irrelevant. Then, a new query is automatically formulated by modifying the previous one and new ranked list is returned, and the process repeats until achieving satisfactory results.

In [Ali *et al.*, 2011], *Ali et al.* propose an approach, called COPARVO, to improve the performance of VSM for recovering traceability links between source code and requirements by partitioning the source code into different sources of information (class names, comments, methods names and class variables). Each information source acts as an expert recommending traceability links established between requirements and code classes using VSM. The top rated experts score each existing link and decide via majority voting if the link is still valid or should be rejected due to evolution in source code, requirements or both.

3.3.1.2 LSI-based Feature Location

In [Marcus et Maletic, 2003a], *Marcus and Maletic* use for the first time LSI for recovering traceability links between documentation and source code. In their approach, source code files without any parsing are used to create LSI's corpus while sections of documentation pages are used as queries. These sections may describe functional requirements, error logs, etc. The experiments prove that the performance of LSI is at least as well as VSM in spite of LSI do not use source code comments. In addition, it requires less preprocessing of the source code and documentation.

In [Marcus *et al.*, 2004], *Marcus et al.* use LSI to link features with their respective source code functions. Maintainers formulate a query describing a feature. Then, LSI is applied to return a list of functions ranked by the relevance to the query. Maintainers inspect this list to decide which ones are actually parts of the feature implementation.

In [Poshyvanyk et Marcus, 2007], *Poshyvanyk and Marcus* propose combining LSI and FCA to link features of a given system to their respective source code methods. LSI is used to return a ranked list of source code methods against each feature. This list may contain a set of irrelevant methods to the feature being located. Therefore, FCA is employed to reduce this list by analyzing the relation between methods and terms that appear in these methods. The top k terms of the first n methods ranked by LSI are used to construct FCA's formal context and create a lattice. The objects of the formal context are methods while the attributes are terms. Concepts in the generated lattice associate terms and methods. Therefore, programmers can focus on the nodes with terms similar to their queries (feature

descriptions) to find feature-relevant methods.

In [Kuhn *et al.*, 2007], *Kuhn et al.* propose an approach to identify *linguistic topics* that reveal the intention of the source code. A *linguistic topic* is a cluster of similar object-oriented source code elements that are grouped together based on their contents. These *linguistic topics* can be composed of different source code granularity (package, class and method). The authors interpret these *linguistic topics* as features implemented by source code. They rely on LSI to compute similarity among a given set of methods, classes or packages. Then, hierarchal clustering is used to cluster similar elements together as *linguistic topics*.

In [Lucia *et al.*, 2008a], *De Lucia et al.* present an incremental LSI-based approach to link requirements with their respective source code classes. Requirements represent queries while class documents represent LSI's corpus. They use a threshold on the similarity value to cut the ranked list of class documents returned by LSI against requirements. In their approach, the similarity threshold value is incrementally decreased to give maintainers control of the number of validated correct classes and the number of discarded false positives classes. This is why their approach is called incremental LSI-based approach. Furthermore, they conduct a comparative study between *one-shot* and *incremental* approaches. In *one-shot* approach, a full ranked list of classes that are textually similar to a requirement is submitted to maintainers for checking. The results prove that the incremental process reduces the effort required to complete a traceability recovery task compared with one-shot approaches.

Lucia et al. [Lucia *et al.*, 2008b] develop a tool called *ADAMS Re-Trace* to recover traceability links between software artifacts of different levels of abstraction by using LSI. Based on this tool, traceability links are recovered by following three steps. In the first step, the software engineer selects artifacts that represent the source (e.g., requirements, sequence diagrams, test cases, code classes). In the second step, he/she selects the artifacts that represent the target. Finally, the software engineer selects the mechanism that is used to show the list of candidate links: threshold based or full ranked list. As a final output, the tool links the source artifacts to their target artifacts.

3.3.1.3 Feature Location by Combining IR with Static Analysis

In [Zhao *et al.*, 2006], *Zhao et al.* use VSM and static analysis to link each feature with their relevant source code functions. Their proposal consists of a two-phase process. In the first phase, VSM is applied to return a ranked list of functions that are similar to a given feature. Then, this list is reduced to a set of feature-specific functions using some algorithm. In the second phase, the authors use a static representation of the source code called Branch-Reserving Call Graph (BRCG). This representation is explored using theses feature-specific functions to identify other relevant functions for these feature-specific functions.

In [Shao et Smith, 2009], *Shao and Smith* combine LSI and static analysis for supporting feature location. First, LSI is applied to rank all methods of a given software system against a given query (feature description). Next, a call graph for each method in the ranked list is created. Next, a method's call graph is investigated to assign it a score. The score represents the number of a method's direct neighbors that are listed in LSI's ranked list. Finally, the cosine similarity from LSI and the scores of call graphs are combined using predefined transformation to produce a new ranked list.

In [Peng *et al.*, 2013], *Peng et al.* propose an iterative context-aware approach to link features with their source code elements (classes and methods). Their approach employs an iterative process with several rounds of feature-to-code mapping. The initial round is based on textual similarity only, and in following rounds textual and structural similarities are combined. Textual similarity between feature descriptions and source code is computed using LSI. The structural similarity between a feature (`f`) and a program element (`P`) was computed by evaluating how much of `f`'s neighboring features and their relations can be mapped to `P`'s neighboring program elements and their relations according to predefined mapping schema.

3.3.1.4 Feature Location by Combining IR with Dynamic Analysis

In [Poshyvanyk *et al.*, 2007], *Poshyvanyk et al.* propose an approach to combine LSI and a dynamic technique called *scenario-based probabilistic ranking (SPR)* to link features with their respective source code methods. Both LSI and SPR return a ranked list of code methods relevant to a feature. Each method in both lists takes a weight expressing the maintainer's confidence about LSI and SPR. Their approach combines both LSI and SPR results to overcome the uncertainty in decision making of feature-to-method linking because methods that are classified as relevant by using LSI can be irrelevant by using SPR.

In [Liu *et al.*, 2007], *Liu et al.* propose a feature location approach called SITIR (SIngle Trace + Information Retrieval). Their approach combines dynamic analysis (i.e., execution traces) and LSI so that for each feature there is only a single execution trace. Then, LSI is used to rank only executed methods in the obtained trace against a given feature instead of ranking all methods in the software system.

In [Asadi *et al.*, 2010], *Asadi et al.* propose to combine LSI, dynamic-analysis and a search-based optimization technique (genetic algorithms) to locate cohesive portions of execution traces that correspond to a feature. Their approach is based on the assumptions that methods implementing a feature are likely to have shared terms and called close to each other in an execution trace. Their approach takes as input a set of scenarios (test cases) that exercise the features. The approach starts by executing the system using these scenarios to collect execution traces so that each trace is a sequence of methods and each feature has one execution trace. Finally, genetic algorithms (GAs) are used to separate methods of an execution trace into segments in order to find the segment that maximizes the cosine similarity between its methods. Similarity between methods is computed using LSI. This is performed by creating a document for each method containing method information (identifiers and comments). Method documents represent at the same time queries and corpus. The similarity between two method document vectors in LSI sub-space is quantified using the cosine similarity. The methods of the obtained segment implement the feature of interest.

In [Eaddy *et al.*, 2008], *Eaddy et al.* propose a feature location approach called *Cerberus*. Their approach uses three types of analysis: dynamic, static and textual. Their approach is the only approach that leverages together all three types of analysis. The core of *Cerberus* is a technique called prune dependency analysis (PDA). By using PDA, a relationship between a program element and a feature exists if the program element should be removed or altered, when that feature is pruned (removed) from the software system. They use the approach proposed by [Poshyvanyk *et al.*, 2007] to combine rankings of program elements from execution traces, with rankings from IR to produce seeds for PDA.

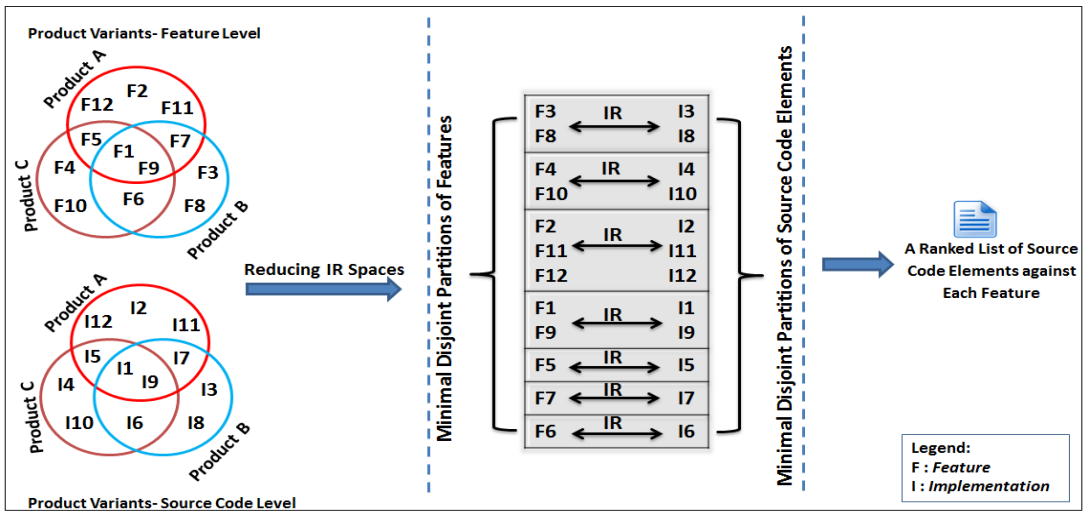

Figure 3.4 : An Example of Feature and Implementation Partitions for IR according to [Xue *et al.*, 2012].

Then, PDA finds additional relevant elements by analyzing different kinds of relationships (such as a method call, inheritance, composition dependency among classes, etc.).

3.3.2 Feature Location in a Collection of Product Variants with IR

Considering product variants together as a set of similar and related products allows the reduction of IR search spaces. This reduction refers to the division of all features and source code elements of given product variants into minimal portions so that this reduction allows to conduct a textual matching between a few features and source code elements that only implement those features. This helps to reduce the number of irrelevant source code elements (false-positive links) returned by IR-based feature location approaches due to similar feature descriptions and implementations. Such a reduction is performed by exploring commonality and variability across product variants at feature and source code levels. Commonality refers to features (resp. their source code elements) that are part of each product while variability refers to features (resp. their source code elements) that are part of one or more (but not all) products.

In the literature, there are only two works that exploit commonality and variability across product variants to support feature location [Xue *et al.*, 2012][Rubin et Chechik, 2012]. One of them uses IR, namely LSI while the other is applicable to any feature location approach.

In [Xue *et al.*, 2012], *Xue et al.* use LSI to link features and their source code units (functions and procedures) in a collection of product variants. They explore commonality and variability of product variants using software differencing tools and FCA to improve the effectiveness of LSI. In their approach, LSI spaces (i.e., feature and source code spaces) are reduced into minimal disjoint partitions of feature and source code units. These partitions represent groups of features (resp. their source code elements) that can not be divided further and also without shared members between these groups. Figure 3.4 shows an example of feature and source code partitions of a collection of

three product variants according to their approach. Each slot in this figure represents a minimal disjoint partition of features and source code elements.

In [Rubin et Chechik, 2012], *Rubin and Chechik* suggest two heuristics to improve the accuracy of feature location approaches when these approaches are used to locate implementations of distinguishing features –those that are present in one product variant while absent in another. Their heuristics are based on identifying a code region that has a high potential to implement distinguishing features. This region is called *set diff* and identified by comparing pair-wisely the code of a variant that provides distinguishing features to one that does not. Figure 3.5 shows an example of all possible partitions (*set diffs*) of features and source code elements obtained by pair-wise comparison of a collection of three product variants according to their approach. In this figure, for example, (B-A) means a set of features (resp. their source code elements) that are present in *B* and absent in *A* ({F3, F8, F6}, {I3, I8, I6}). Although their approach helps to reduce IR's search spaces, the achieved reduction is not minimal. For example, F7 (resp. its code elements) is grouped with other features (resp. their code elements) while F7 (resp. its code elements) in Xue et al.'s work is grouped alone (see Figure 3.4). Additionally, their work does not pay attention to how code portion that implements shared features among all product variants can be obtained.

In the literature, there are two works [Ziadi *et al.*, 2012][Al-Msie'Deen *et al.*, 2013] similar to some extent to approaches mentioned in this category, in spite of the fact that they do not use feature descriptions and IR for determining feature implementations. We consider these works because they exploit what product variants have in common at the source code level to identify feature implementations. These works are as follows:

In [Ziadi *et al.*, 2012], Ziadi et al. propose an approach to identify portions of source code elements that potentially may implement features in a collection of product variants. They exploit what product variants have in common at the source code level by performing several rounds of intersections among source code elements of product variants. In the first round, the source code elements shared between all product variants are obtained. In the next rounds, source code elements shared among some product variants are obtained. The result of each intersection may potentially represent feature implementation(s). According to their approach, they consider source code elements (packages, classes, methods and attributes) that are shared between all product variants as implementation of a single feature. However, this implementation may correspond to more than one feature when all product variants share two features or more. Moreover, their approach does not distinguish the implementation of features that always appear together, such as AND-Group of features.

In [Al-Msie'Deen *et al.*, 2013], Al-Msie'Deen et al. propose an approach similar to Ziadi et al.'s approach. Their approach exploits common source code elements across product variants to identify segments of source code elements, which potentially may implement features. They rely on FCA for conducting several intersections between source code elements of product variants to identify these segments. The formal context is defined as follows: objects represent names of product variants, attributes represent all unique source code elements of product variants and the relation between objects and attributes refers to which products possess which source code elements. Based on their approach, the intent of each concept in the generated lattice may correspond to the implementation of one or more features. Therefore, they rely on LSI and again on FCA to divide intent of each concept into sub-segments of textual similar source code elements. The idea behind this division is that code

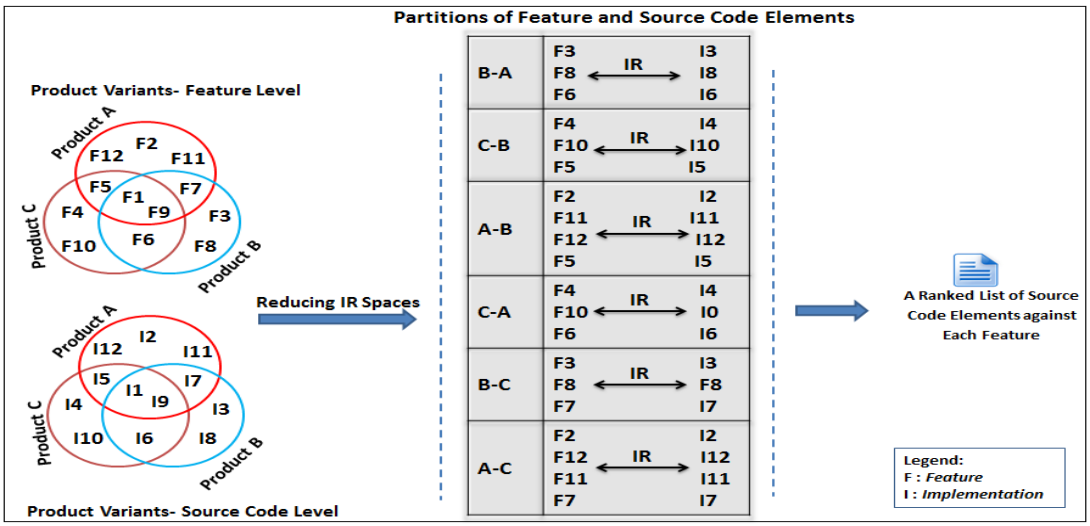

Figure 3.5 : An Example of Feature and Implementation Partitions for IR According to [Rubin et Chechik, 2012].

elements that collaborate to implement a feature share similar terms, and hence they can be grouped together as a feature implementation. However, it is not necessary that the intent of each concept represent implementation of feature(s) because source code elements that are shared between two or more features appear as a separated concept. This leads to identify features more than the features actually provided by a given collection of product variants, and hence missing source code elements that are relevant to features that actually are provided by this collection.

3.4 Evaluation of IR-based Feature Location Approaches

In this section, we present the evaluation criteria of IR-based feature location approaches. Also, we compare surveyed approaches according to these criteria.

3.4.1 Evaluation Criteria

As we contribute to improve the IR-based feature location in a collection of product variants, we propose a set of criteria related to this contribution. We mainly organized these criteria into two categories: (i) criteria related to strategies used to reduce IR search spaces, and (ii) criteria related to information used to improve IR-based feature location.

- **Criteria related to strategies used to reduce IR search spaces**. This category consists of two criteria as follows:

  1. **Awareness of commonality and variability**: whether the studied work consider commonality and variability across software product variants to reduce IR spaces. As mentioned

earlier, this reduction improves the performance of IR for locating feature implementations.

2. **Awareness of abstraction gap between feature and source code levels**: whether the studied work considers the big abstraction gap between feature and source levels, as they belong to different levels of abstraction. The basic idea behind this criterion is as a follows. Terms of feature descriptions may be scattered over a large number of source code elements (e.g., classes, methods), which may hinder finding textual matching between features and their implementing source code elements. This scattering is due to the fact that a feature's implementation spans multiple source code elements (e.g., classes, methods, etc.). We assume that source code elements that contribute to implementing a feature have shared terms and they also are invoked near to each other. We believe that by grouping together these source code elements as a cluster(s), we may obtain a better textual matching with a given feature description, than considering each source code element individually. Such clusters represent an intermediate level between feature and source code levels. Also, such clusters reduce the source code space of IR by grouping source code elements that collaborate to support the same function (functional requirement) together and linking this group as a single unit to the same feature that provides that function. This leads to a reduction in the number of source code elements that should be linked to features, and hence decreases the opportunity of establishing false-positive links between features and individual source code elements.

- **Criteria related to information used to improve IR-based feature location.** This category consists of two criteria as follows:

  1. **Using static source code information**: whether the studied work uses static source code information, such as control dependency and data flow for supporting feature location. Static analysis represents and additional source of information for supporting feature location process.
  2. **Using textual information describing features**: whether the studied work exploits available feature information, such as feature description to locate a feature implementation. Feature description is like a guide for IR during the source code to locate the correct implementation of given feature(s).

- **Using well-known metrics for evaluation**: whether the studied work evaluate the obtained results using the well-known metrics of IR, which are *precision*, *recall* and *F-measure*. The way in which a feature location approach is evaluated provides researchers with useful information about the approach quality and robustness.

3.4.2 Evaluation

In this section, we compare IR-based feature location approaches. Table 3.1 summarizes the studied approaches according to the evaluation criteria.

All approaches that belong to the first category of IR-feature location (*feature location in single software product with IR*) do not consider commonality and variability across product variants. This

Table 3.1 : Summary of IR-feature location approaches.

	Studied Work	Awareness of commonality and variability	Awareness of abstraction gap between feature and source code levels	Using static source code information	Using textual information describing features	Using well-known metrics for evaluation	Source code granularity	Technique used
Single Software Product	[Ali *et al.*, 2011]	no	no	no	yes	yes	class	VSM
	[Poshyvanyk et Marcus, 2007]	no	no	no	yes	no	method	LSI, FCA
	[Zhao *et al.*, 2006]	no	no	yes	yes	yes	function	LSI, static analysis
	[Marcus *et al.*, 2004]	no	no	no	yes	yes	function	LSI
	[Lucia *et al.*, 2008a]	no	no	no	yes	yes	class	LSI
	[Peng *et al.*, 2013]	no	no	yes	yes	yes	class, method	LSI, static analysis
	[Liu *et al.*, 2007]	no	no	no	yes	no	method	LSI, dynamic analysis
	[Shao et Smith, 2009]	no	no	yes	yes	yes	method	LSI, static analysis
	[Gay *et al.*, 2009]	no	no	no	yes	no	method	VSM
	[Poshyvanyk *et al.*, 2007]	no	no	no	yes	no	method	LSI, dynamic analysis
	[Eaddy *et al.*, 2008]	no	no	yes	yes	yes	method, attribute	LSI, dynamic analysis, static analysis
	[Asadi *et al.*, 2010]	no	no	no	no	yes	method	LSI, dynamic analysis, genetic algorithm
	[Kuhn *et al.*, 2007]	no	no	no	no	no	package, class, method	LSI, hierarchal clustering
	[Antoniol *et al.*, 2002]	no	no	no	yes	yes	class	VSM, probabilistic model
	[Marcus et Maletic, 2003a]	no	no	no	yes	yes	class	LSI
	[Lucia *et al.*, 2008b]	no	no	no	yes	yes	class	LSI
Product Variants	[Xue *et al.*, 2012]	yes	no	no	yes	yes	function	LSI
	[Rubin et Chechik, 2012]	yes-partially	no	no	no	yes	code region (class, method, attributes, function, procedure)	own algorithm
	[Al-Msie'Deen *et al.*, 2013]	yes-partially	no	no	no	yes	package, class, method, attributes	LSI, FCA
	[Ziadi *et al.*, 2012]	yes-partially	no	no	no	no	package, class, method, attributes	own algorithm

is because these approaches are designed to deal with product variants as individual and unrelated entities. For approaches that belong to the second category (*feature location in a collection of product variants*), the works of Ziadi et al. [Ziadi *et al.*, 2012] and Al-Msie'Deen et al. [Al-Msie'Deen *et al.*, 2013] only consider reducing source code space of product variants by only exploiting what these variants have in common at the source code level. Therefore, the achieved reducing is not minimal because their approaches do not consider differences at source code level among product variants. The work of *Rubin and Chechik* [Rubin et Chechik, 2012] partially reduces the feature and source code spaces of product variants, as their approach considers only differences and ignores commonality across product variants. The work of *Xue et al.* [Xue *et al.*, 2012] reduces feature and source code spaces of product variants into minimal disjoint partitions by considering both what these variants have in common and how they differ.

Table 3.1 shows that none of the surveyed approaches consider the abstraction between feature and source code levels. It also shows that most surveyed approaches use feature information as input of feature location process except the works of Kuhn et al. [Kuhn *et al.*, 2007], Asadi et al. [Asadi *et al.*, 2010], Ziadi et al. [Ziadi *et al.*, 2012] and Al-Msie'Deen et al. [Al-Msie'Deen *et al.*, 2013], as these works support feature identification.

Table 3.1 shows that most surveyed approaches use the well-known metrics (*precision*, *recall* and F-measure) for evaluation IR results. However, the works of (*Ziadi et al.* [Ziadi *et al.*, 2012], *Poshyvanyk and Marcus* [Poshyvanyk et Marcus, 2007], *Liu et al.* [Liu *et al.*, 2007], *Gay et al.* [Gay *et al.*, 2009], *Poshyvanyk et al.* [Poshyvanyk *et al.*, 2007] and *Kuhn et al.* [Kuhn *et al.*, 2007]) use specific metrics for evaluation. Also, the Table 3.1 shows that few works use static source code information for supporting the IR-based feature location process and all these works belong to the first category of IR-feature location. Additionally, it shows that the surveyed approaches deal with different source code granularities that could be linked to features (classes, methods, functions and a combination of classes, methods and attributes).

3.5 Conclusion

In this chapter, we presented state-of-the-art information about feature location, namely IR-based feature location. Feature location approaches are classified into three classes based on the type of analysis used: *dynamic, static* and *textual*. IR-based feature location approaches belong to the textual class. We classified IR-based feature location approaches based on how they deal with product variants into two categories: *feature location in single software* and *feature location in a collection of product variants*. All approaches of the first category are designed for locating feature implementations in single software product, and hence they can not be able to exploit commonality and variability to reduce features and source code space of IR. These approaches represent the conventional application of IR. Few approaches are proposed in the second category, which deals with product variants as a family of similar and related products. Dealing with product variants in such a way provide additional input to the feature location process, and hence improves the performance of IR-feature location compared with the conventional application. This input represents commonality and variability distribution across product variants at the feature and source code levels. This input helps to reduce IR search spaces (feature and code spaces). Only one of these approaches (*Xue et al.* [Xue *et al.*, 2012]) reduces these spaces into minimal disjoint partitions while others achieve non-

minimal reduction of these spaces and some of them only consider reducing source code space, as we have seen. We started to work in parallel with [Xue *et al.*, 2012] on improving the IR-based feature location in a collection of product variants by reducing the IR search spaces. None of the surveyed approaches consider the abstraction gap between feature and source code levels. Additionally, none of the approaches of the second category consider static source code information during the feature location process and some of them do not consider information related to features (feature descriptions).

Based on our analysis of state-of-the-art related to IR-based feature location, none of the surveyed approaches meet all criteria presented in this chapter. For this, we propose an approach effectively analyzes commonality and variability in product variants, which leads to reducing the search spaces of IR into minimal disjoint sets. Also, our approach considers together other factors to improve the effectiveness of IR for locating feature implementations, which include bridging the abstraction gap between feature and source code levels, using static source code information and feature descriptions.

CHAPTER 4

Applications of Feature Location: Feature-Level CIA and Supporting Reverse-Engineering SPLA

Preamble

In this chapter, we present a state-of-the-art application of feature location for feature-level Change Impact Analysis (CIA) and supporting reverse engineering Software Product Line Architecture (SPLA) from product variants. Section 4.1 deals with feature-level CIA. During this section, we present the main concepts of CIA and a comparison of the CIA approaches, following a set of criteria that we consider relevant to our contribution. Section 4.2 is dedicated to software product line architecture (SPLA). In this section, we present the main concept in SPLA. Then, we survey the approaches related to reverse engineering SPLA and conduct a comparison based on a set of criteria that are relevant to our contribution. Section 4.3 concludes this chapter by showing the need to propose approaches for both feature-level CIA and reverse engineering SPLA.

4.1 Feature Level Change Impact Analysis

As feature-level CIA is one of the applications of feature location in our work, we study in this section state-of-the-art CIA. In the beginning, we present the main concepts in the CIA process. Next, we review studied works. Finally, we evaluate these works following a set of criteria.

4.1.1 Change Impact Analysis: Main Concepts

Change impact analysis (CIA) is defined as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [Bixin *et al.*, 2012][Arnold, 1996]. Changes made to one part of a software system may cause unpredictable and undesirable effects on other parts of the same software, called *ripple effects* [Bixin *et al.*, 2012]. CIA approaches determine the ripple effects of a given change proposal. They take as input the changes that the maintainer plans to achieve (called the *change set*) and determine which other parts could be affected, called the *estimated impact set* (EIS). The EIS set also includes the members of *change set*.

Figure 4.1 shows the whole CIA process [Bohner, 2002][De Lucia *et al.*, 2008]. This process starts by analyzing both the change request specifications and the system code to identify the *change set* (e.g., source code, requirements, use cases, test cases, etc.). Artifacts in the change set are then analyzed to identify other artifacts estimated to be affected (*EIS*). When the change is implemented, the *Actual Impact Set (AIS)* is discovered. AIS is the set of artifacts actually modified. As shown in Figure 4.1, CIA is an iterative process. During the implementation of a change request, new affected artifacts that are not included in the EIS may be discovered. The set of such artifacts is called *false negative impact set*. Moreover, it is also possible that some artifacts in the EIS are not impacted by the change being implemented. The set of these artifacts is called the *false positive impact set*.

4.1.2 CIA Approaches: Classification and Presentation

Bohner and Arnold describe two categories of CIA approaches [Bohner, 2002]: *traceability-based CIA* and *dependency-based CIA*.

- **Traceability-based CIA:** refers to approaches that perform CIA between software artifacts belonging to the same level of abstraction and the corresponding artifacts at other abstraction levels by exploiting traceability between them (e.g., code-to-design). Figure 4.2 shows an example of traceability links between different levels of abstraction.
- **Dependency-based CIA:** refers to approaches that perform CIA among software artifacts belonging to the same level of abstraction (e.g., between source code elements). This is performed by analyzing dependency between considered artifacts.

Feature-level change impact analysis (CIA) is the process of determining the affected features for a given change proposal before a change is implemented. We are interested in traceability-based CIA as feature-level CIA belongs to this category. In the following, we detail the approaches that belong to this category. For dependency-based CIA, we only give a synthesis of existing work that belongs to this category.

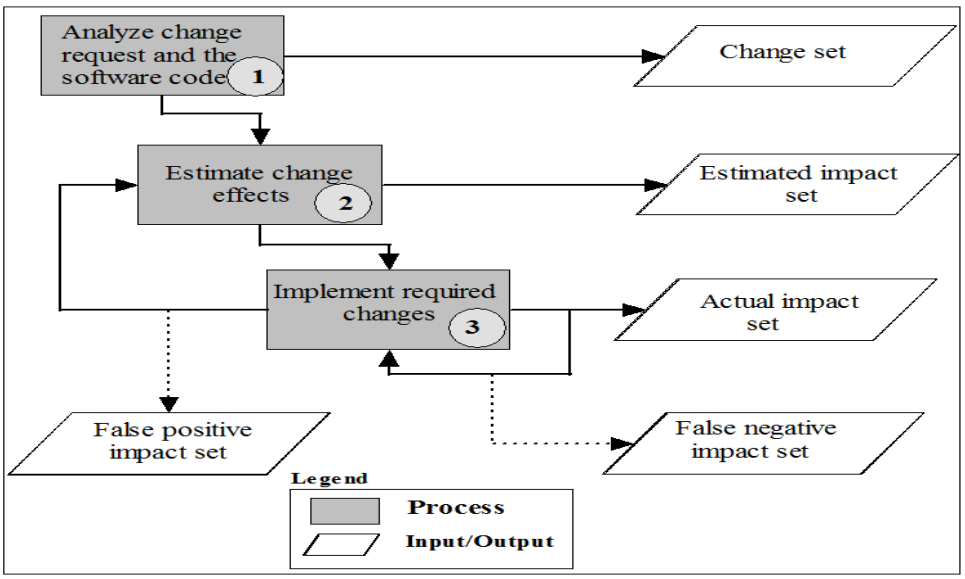

Figure 4.1 : Change Impact Analysis Process [Bixin *et al.*, 2012].

4.1.2.1 Traceability-based CIA

When a software system is changed, the impact of change is not limited to a certain type of software artifacts but goes through artifacts of all software life cycle phases. A survey about traceability-based CIA approaches is proposed by De Lucia et al. [De Lucia *et al.*, 2008]. They analyzed the role of traceability relations in the impact analysis. However they do not study and compare specific traceability-based CIA approaches.Below, we present the studied work that supports CIA across different levels of abstraction.

In [Revelle *et al.*, 2011], *Revelle et al.* proposed a feature coupling metric based on source code information: structural (e.g., method calls) and textual (e.g., comments and identifiers) information. In their approach, feature coupling metric is composed of textual feature coupling and structural feature coupling. The former is measured by computing textual similarity between two methods as well as between a method and a given feature. The later is measured as the ratio of the number of methods shared by the features to the total number of methods associated with the two features. They assumed that traceability links exist between features and source code elements that implement those features. In their work, feature coupling was used to perform CIA. It is the degree to which the source code elements of a feature (e.g., methods, attributes, classes) depends on elements outside the feature [Apel et Beyer, 2011]. The idea behind feature coupling is that features that are strongly coupled to a feature being modified are the most likely to be affected. According to their approach, the coupled features can be determined by setting a threshold value for coupling strength, since features with a coupling value equal to or above a given threshold are considered as coupled features to the feature being modified.

Figure 4.3 shows the architecture of the feature coupling component proposed by[Revelle *et al.*, 2011]. First, the source code of a system is parsed into methods. Then, the text of the methods is

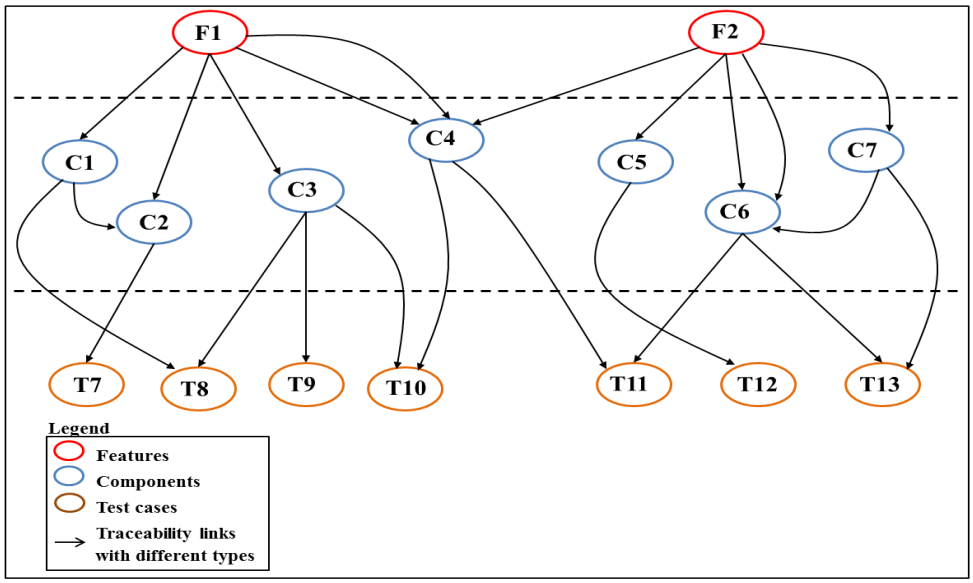

Figure 4.2 : Traceability links between different levels of abstraction represented as multigraph.

preprocessed to form the documents of the corpus. Pre-processing always removes stop words and programming language keywords and splits compound identifiers. Options include removing comments from the corpus and performing stemming. Then, LSI is used to create a word-by-document matrix that describes the distribution of terms in the methods of the corpus. Through the use of SVD, a semantic subspace is constructed in which each method from the corpus is represented as a vector. The cosine similarity between two vectors is a measure of the textual similarity between two methods. Given the similarities between methods and the mappings of features to methods, the proposed approach can compute textual feature coupling. To compute structural feature coupling, the tool simply requires feature-method maps as well as dependency information.

In [Ibrahim *et al.*, 2005], *Ibrahim et al.* present an approach for CIA of object oriented applications. The CIA is performed at requirements, design, source code and test case levels by connecting artifacts of these levels. This connection enables a comprehensive impact analysis as proposed by the authors. In their approach, packages and classes are considered as design elements while methods are considered as source code elements. They gather traceability relations from different sources. Requirements and test cases are linked by using system documentation. Test cases and methods are linked via test execution. Methods and classes are linked by static program analysis. In their approach, the change may occur at different levels of abstraction and traceability is used to study impact analysis at these levels. Figure 4.4 shows a traceability from the point of view of requirements. For example, R1 is a requirement that has direct impacts on test cases T1 and T2. R1 also has direct impacts on the design D1, D2, D3 and on the code component C1, C3, C4. Meanwhile T1 has its own direct impact on D1 and D1 on C4, C6, etc which reflect the indirect impacts to R1. The same principle also applies to R2. R1 and R2 might have an impact on the same artifacts e.g., on T2, D3, C4, etc.

In [Xiao *et al.*, 2007], *Xiao et al.* propose an approach to study CIA at source code level for changes

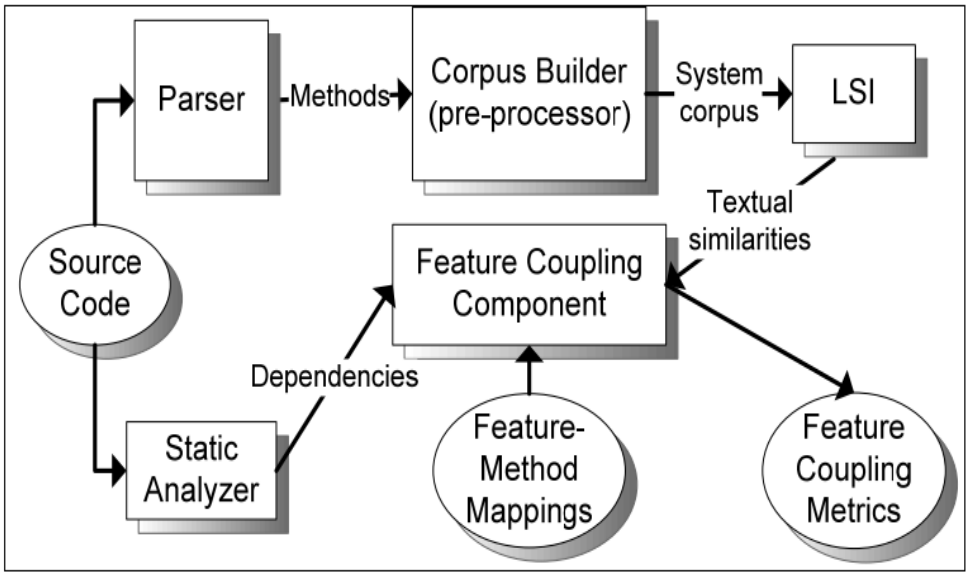

Figure 4.3 : Architecture of the feature coupling component [Revelle *et al.*, 2011].

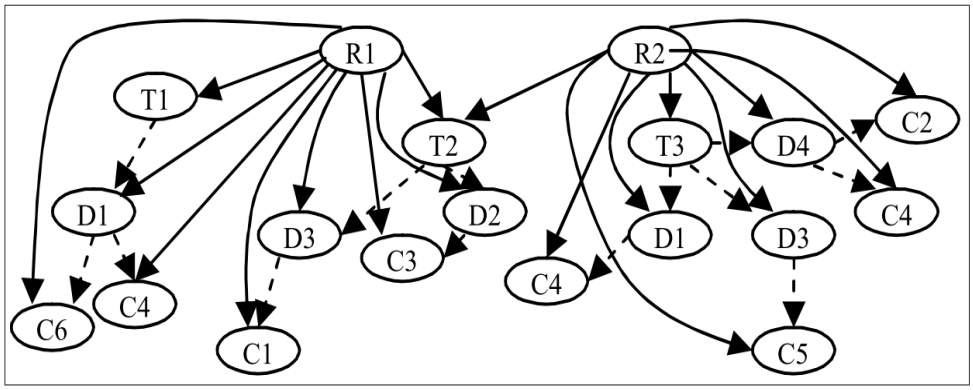

Figure 4.4 : Traceability from the requirement traceability perspective [Ibrahim *et al.*, 2005].

induced at the business level in service-oriented business applications. For a given change request to modify the implementation of a business process such as,*online money order transferring*), their approach studies the impact of the change at both business and source code levels at the same time. Firstly, it identifies the elements of other business processes that may require a modification to implement the change request. Then, it uses existing traceability links between source code elements and business process elements that are either changed or impacted in order to find their corresponding source code elements. Finally, these identified elements are used as staring point to find other code elements that could be modified by using the call graph.

In [Briand *et al.*, 2009], *Briand et al.* propose an approach for supporting the selection of regression test cases based on impact analysis of software architectures modeled in UML. Regression testing

is used only to verify the correctness of changes implemented because executing all test cases after each change is not feasible in practice. The change in their work occurs at the design level to create a new design for a new version of a current software system. They assume that there are traceability links between design elements and test cases. These links propagate change impact to test level and hence they can be used to estimate regression testing effort which is usually substantial [Leung et White, 1991]. Using these links, test cases are classified into: obsolete, re-testable and reusable. Obsolete is a test case that cannot be executed on a new version of the system as it is invalid in the new version. Re-testable is a test case that is still valid but needs to be rerun for the regression testing. Reusable is a test case that is still valid and does not need to rerun to ensure regression testing. In their work, the impact estimation and the selection of test cases is based on identifying differences between two versions of the system architecture.

In [Bo Yu, 2004], *Yu et al.* propose the concept of *requirements change probability* to estimate whether a change to one component, due to requirements evolution will spread to other components. *Requirements propagation probability* is the probability that a change made to a component (A) causes a change to a neighboring component (B). To compute this probability, firstly, architectural components of a given system are extracted and organized into a N × N matrix, called *requirements propagation probability*. This matrix is used to store the propagation probabilities between pairs of components. Secondly, the probabilities are computed by applying three matrices on each component: *backward functional call dependency*, *forward functional call dependency* and *total functional call dependency*. Those matrices with size N-1 × N-1 and the semantic of their values are as follows respectively: how many functions of component (i) are called by other components (j); how many functions from other components (j) are called by a component (i); combining the previous two. The semantic of each row in *requirements propagation probability* matrix is as a follows: if the row corresponding to a component *A* has higher values, we deduce that changes to this component (A) must be avoided because they propagate widely throughout the system. Impacted components can then be determined by applying a probability threshold.

In [Hammad *et al.*, 2011], *Hammad et al.* propose an approach to automatically determine if a given source code change impacts the design (i.e., UML class diagram) of a system. Their goal is to keep the architecture synchronized with the source code. They distinguish between code changes that impact a system architecture (such as adding/deleting a method or class), and those that do not (such as changing a control loop). For a given code change causing two versions of the source code, firstly their approach transforms source code of the two versions into XML format by using *srcML* tool in order to support the static analysis required. Secondly, the differences between the two versions are determined by applying *srcDiff* tool to the XML output. Finally, the changes that impact the design are identified from the code differences via a number of queries. For example, if the results of a query detect a source code class which is not represented in design models, this means that there is an impact on the architecture and a design change took place.

In [Khan. Simon Lock, 2009], *Khan* utilizes dependencies between requirement-level concerns and architectural components to study the impact of requirement changes at the architecture level. They investigate whether dependencies between them help to identify unstable components and anticipate changes. Taxonomy of dependencies is established as the basis of their CIA approach. Dependencies are divided based on their nature into *goal, task, service, conditional, infrastructural and usability*. Also, they classify the granularity of a dependency into: *overlap, intertwine* and *conform*.

For example, overlap dependency holds when a set of requirements belonging to different concerns are linked to the same operation of an architectural component. These granularities are used as predictors for determining architectural components that are more receptive to change made to requirements.

In [Chechik *et al.*, 2009], *Chechik et al.* propose a model-based approach for propagating changes between requirements and design models. Their approach propagates changes between requirements-level activity diagrams, and design-level sequence diagrams. A change propagation algorithm is proposed to identify and localize the effects of requirement changes at design level.

4.1.2.2 Dependency-based CIA

Most of dependency-based CIA approaches are code-based CIA (i.e., change impact analysis at source code level). A survey about these approaches can be found in [Bixin *et al.*, 2012]. According to [Steffen, 2011][Bixin *et al.*, 2012], there are many approaches proposed for supporting CIA at source code level. Below, we give an overview of these approaches by presenting them based on the technique used.

1. **Call Graphs:** the methods/procedures that are changed may impact other source code methods/procures directly or indirectly. Therefore, analyzing the call-behavior of system's methods/procedures can help to assess the impact of changed methods/procedures. By statically analyzing the source code, method calls are extracted and stored in a graph or matrix. Then, this graph is used by maintainers to estimate the propagation of a given change [Ryder et Tip, 2001][Xia et Srikanth, 2004] [Badri *et al.*, 2005].

2. **Dependency Analysis:** there are several types of dependencies between source code elements, such as composition and inheritance dependencies. They can be extracted by static source code analysis. These dependencies can be used to estimate the change propagation between source code elements. [Briand *et al.*, 1999][Kung *et al.*, 1994][Rajlich, 1997][Zalewski et Schupp, 2006][Petrenko et Rajlich, 2009][Black, 2001][Li et Offutt, 1996][Jász *et al.*, 2008][Gwizdala *et al.*, 2003][Pirklbauer *et al.*, 2010][Hoffman, 2003].

3. **Program Slicing:** is the computation of the set of programs statements (i.e., the program slice) that may affect the value of a variable. Slicing removes all code statements which are irrelevant to the slicing criterion, i.e. which do not affect the state of a variable and thereby being of no use for impact analysis. Slicing is performed by statistical source code analysis [Gallagher et Lyle, 1991][Hutchins et Gallagher, 1998][Tonella, 2003][Binkley et Harman, 2005][Vidács *et al.*, 2007].

4. **Dynamic Analysis:** allows producing execution traces containing only methods which have been invoked during the execution of a program. They allow estimation of the impact of a method change by determining which methods were called after the changed method, thus being possibly impacted, too [Law et Rothermel, 2003b][Law et Rothermel, 2003a][Orso *et al.*, 2003][Breech *et al.*, 2006][Apiwattanapong *et al.*, 2005][Beszédes *et al.*, 2007][Vanciu et Rajlich, 2010].

5. **Information Retrieval (IR):** IR techniques exploit textual source code information (e.g., identifiers) to find similar terms in different source code elements (e.g., classes, methods) in order

to establish a relation between these elements. This relation can be used to study the change propagation among source code elements [Antoniol *et al.*, 2000][Poshyvanyk *et al.*, 2009].

6. **History Mining:** is a technique related to mining software repository (MSR) [Kagdi *et al.*, 2007]. It identifies clusters or patterns of entities from software repositories which are often changed together so that a change made to one entity of a cluster is likely to affect all the other members within that cluster as well [Fluri *et al.*, 2005][Fluri et Gall, 2006][Gall *et al.*, 2003][Zimmermann *et al.*, 2005][Gîrba *et al.*, 2005][Gîrba et Ducasse, 2006][Robbes07, 2007][Gîrba *et al.*, 2007][Ying *et al.*, 2004][Bouktif *et al.*, 2006][Zimmermann et Weißgerber, 2004][Robbes *et al.*, 2007].

4.1.3 Evaluation of Traceability-based CIA

4.1.3.1 Evaluation Criteria

We consider a set of criteria for evaluating the existing traceability-based CIA approaches that support change management from a SPL manager's point of view. This view involves studying the impact of a change at the feature level and providing quantitative measures to help SPL's manager for making decisions.

- **Abstraction level considered for performing CIA**: this criterion aims to determine the level of abstraction that CIA process supports. By evaluating the studied works according to this criterion, we can explore the approaches that perform CIA at the feature level.
- **Quantitative measures for evaluation of the change impact:** this criterion aims to determine approaches that provide quantitative measures to evaluate the impact of a given change. Such measures help stakeholders relevant to the abstraction level considered by CIA process to easily make a decision either committing the change or finding another solution to implement a change. This criterion is divided into two criteria:
  - **The impact degree of EIS members**: whether the studied work computes to which degree the EIS members are impacted. Based on this criterion, we can determine and evaluate which feature-level CIA approach can compute the impact degree of each affected feature. This is because SPL's manager may be interested to know the impact degree of specific features for their importance.
  - **The changeability of a whole system**: whether the studied work estimates the changeability of a whole system. Based on this criterion, we can determine and evaluate which feature-level CIA approach provides quantitative measure to compute the percentage of affected features for a given change proposal (i.e. the ratio of affected features to all features.). Such a changeability is classified into low, medium or high [Sun et Li, 2011]. Therefore, SPL's manager can select appropriate modification strategy according to the changeability level as a change can be implemented in different ways.
- **Using efficient metrics for evaluation**: whether the studied work uses metrics to quantify elements in the EIS that are not impacted (*false-positive*), and evaluates the elements that are not identified but are impacted (*false-negative*). The way in which CIA approach is evaluated provides information about the approach quality and robustness.

Table 4.1 : Summary of traceability-based CIA approaches.

Studied Work	Abstraction level considered for performing CIA	Quantitative measures for evaluation of the change impact		Using efficient metrics for evaluation
		The impact degree of EIS members	The changeability of a whole system	
[Ibrahim *et al.*, 2005]	requirement	no	no	no
[Xiao *et al.*, 2007]	source code	no	no	no
[Briand *et al.*, 2009]	architecture	no	yes	no
[Bo Yu, 2004]	architecture	yes	no	no
[Revelle *et al.*, 2011]	Feature	no	no	yes
[Hammad *et al.*, 2011]	design (class digram)	no	yes	no
[Chechik *et al.*, 2009]	design (sequence diagram)	no	no	no
[Khan. Simon Lock, 2009]	architecture	no	no	no

4.1.3.2 Evaluation

Table 4.1 summarizes the studied approaches according to the evaluation criteria. Below, we evaluate our review of CIA approaches that belong only to the traceability-based CIA category following the proposed criteria.

Among all studied approaches, there is only one approach that supports feature-level CIA for source code changes (see Table 4.1) [Revelle *et al.*, 2011]. As we mentioned before, the approach of Revelle et al. [Revelle *et al.*, 2011] considers that features (resp. their implementations) that are strongly coupled to the feature being modified are the most likely to be affected. The remaining works perform CIA at other levels of abstraction (requirement, architecture, etc.)

Most studied approaches do not pay attention to compute the impact degree of each member of the EIS, as these approaches only concerned with finding a list of affected elements. The approach of Yu et al. [Bo Yu, 2004] is the only one that computes the impact degree of EIS members that represent architectural components. In their approach, the impact probability of each component due to changes made to requirements is computed.

Most studied approaches do not consider the changeability of the whole system considered against a given change request except the work of Briand et al. [Briand *et al.*, 2009] and Hammad et al. [Hammad *et al.*, 2011]. The work of Briand et al. estimates the regression testing efforts due to changes made to software architectures through classifying the test cases of a given system into three groups, as we have seen. The work of Hammad et al. determines the number of design changes of the whole UML class diagram of a given system, due to changes made to source code level.

The work of Revelle et al. [Revelle *et al.*, 2011] is the only one that uses efficient metrics to evaluate

the EIS. These metrics are inspired from IR: *precision and recall.* Precision measures the accuracy of the EIS according to the actual impact set. Recall measures to what degree the EIS covers the actual impact set. Based on these definitions, we believe that these metrics are efficient because they help to quantify elements in the EIS that are not impacted and the elements that are not identified but are impacted. All other studied approaches assume that the computed EIS is safe in terms of not excluding any entity that is actually impacted without any confirmation. Considering that these approaches use static and dynamic analyses that may return false-positive and false-negative entities [Bogdan *et al.*, 2013].

4.2 Software Product Line Architecture Development: Feature-to-Architecture Traceability

In this section, we present the main concepts in SPLA and study research works related to building Software Product Line Architecture (SPLA).

4.2.1 Software Product Line Architecture: Main Concepts

According to Pohl et al. [Pohl *et al.*, 2010], software product line architecture is a core architecture that captures the high level design for the products of the SPL, including the variation points and variants documented in the variability model (feature model). These decisions concern the organization of components and general rules that these components have to obey. A component is a well-known building unit to build an architectural view of a software system. A software component is a unit of composition with contractually specified interfaces, explicit context dependencies only, subject to composition by third parties and also it can be deployed independently. In object-oriented applications, a component can be considered as a group of classes collaborating to provide a function of the application [Allier *et al.*, 2011]. Components in SPLA are organized into *mandatory components* that are part of each architecture of SPL's products, and a set of *variation points* (VPs) that represent switching points where the SPLA offers different variants (i.e., components) to choose from [Pohl *et al.*, 2010][Zhang *et al.*, 2008]. Architectural components for all SPL's products are derived from SPLA by making a choice at each variation point to select the appropriate component(s) taking into account the rules defined in SPLA.

Globally, commonality and variability in SPLA (i.e., mandatory components and VPs of components) originate from commonality and variability of customers' requirements/features [Pohl *et al.*, 2010]. As mentioned earlier, variability in customer's requirements is represented by feature groups in the feature model (FM) so that customers can choose feature(s) from each group depending on the customer's needs. Each group represents a VP of features. This VP is reflected in SPLA as a VP of components. Therefore, to utilize the potential for variation in customer requirements/features to build SPLA, an explicit link is needed between features where variation can occur (VPs of features) and the places in the SPLA architecture (VPs of components) that are designed to support those VPs of features. An example to clarify the concept of VP of features and components is shown in Figure 4.5. This figure shows all available alternatives for a customer to choose a phone that only supports *color*, *high resolution* or *basic* screens. It shows that each screen option is realized by a combination consisting of two components, and the appropriate combination is based on customer choice. Therefore, selecting the appropriate combination of components involves an explicit link between VPs of features and VPs of components. These links bind variability in SPLA (VPs) according to customers' needs.

Different terms have been used to refer to architecture in SPLE, such as *software product line architecture*, *domain architecture*, *domain-specific architecture*, *configuration architecture* and sometimes called *reference architecture* [Nakagawa *et al.*, 2011]. In this thesis, we use the term software product line architecture or SPLA for short.

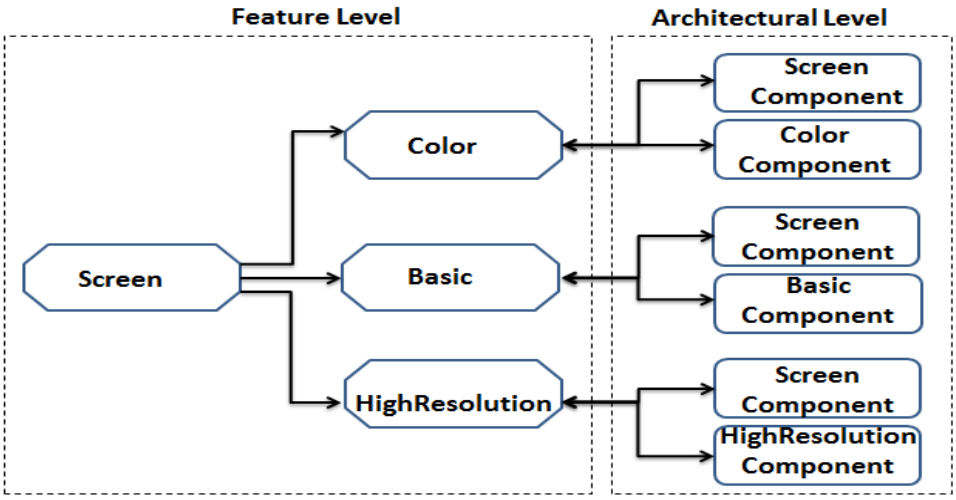

Figure 4.5 : An Example of a Variation Point at the Feature and Architecture Levels.

4.2.2 Presentation of SPLA Engineering Approaches

Although SPLA is a key asset in SPL core assets and building such architecture is a costly task, reverse engineering SPLA from software product variants is not considered in the literature. Below, we present approaches that provide mapping between features and architectural components in order to build SPLA into SPLE context (i.e., forward engineering way for SPLA development).

In [Liu et Mei, 2003], *Liu and Mei* propose the notion of a natural mapping between a FM and architecture. The natural mapping refers to the fact that each feature should be linked to component(s)/subsystem(s). They propose a feature–oriented requirements modeling process to build a FM from a set of relevant detailed requirements. Then, the mapping between features and components is performed manually without any details about how this mapping should be executed.

In [Sochos *et al.*, 2006], *Sochos et al.* propose creating SPLA based on FM. In their approach (called Feature-Architecture Mapping, for short FArM), a strong mapping between features and components is established based on four transformations of the initial FM leading to SPLA. During the transformation process, feature tangling (i.e. when two or more features are implemented by a single component) and feature scattering (i.e. when two or more components implement a single feature) are minimized. The transformed FM contains only functional features. According to their approach, components are developed from scratch to implement the transformed features. The interactions among components is based on feature interactions (relations between features in the FM).

In [Trinidad *et al.*, 2007], *Trinidad et al.* propose automatically building a component model from a FM for developing dynamic SPL. The main problem that they address is to build an architecture that dynamically adapts itself to changing requirements. They propose respectively mapping mandatory and optional features to mandatory and optional components. They create for each feature a component and relations among features become relations among components.

In [Zhang *et al.*, 2008], *Zhang et al.* propose mapping feature to architectural components for

building SPLA. Features are extracted from FM. In their approach, features are classified according to the variability type into *mandatory* and *optional*. Also, mandatory and optional features are further classified according to the impact of their implementation on each other into non-crosscutting and crosscutting features. In their approach, a component is created for each feature. The components of crosscutting features are implemented by object-oriented techniques while the components of non-crosscutting features are implemented by aspect-oriented techniques.

In [Diana L. et Hassan, 2004], *Diana and Hassan* propose an approach called the Variation Point Model (VPM), which models variability at the architecture level, beginning with requirements. According to this model, variability is modeled as variation points. The concept of variation point support four views: requirements, component, static and dynamic views. The requirements variation point view shows the variation point in terms of the requirements. Such a view takes an enumerated statement form. The component variation point view is a composite view that shows all the variation points that belong to a particular component. This view also shows if a variation point belongs to more than one component and if several variation points belong to one component. In this model, each variation point at the requirement level is traced to one or more components. The static variation point view shows which classes, methods or attributes constitute the component variation point. The dynamic variation point view shows the interaction needed between components.

In [Bachmann et Bass, 2001], Bachmann et al. presents experience with explicitly managing variability within a software architecture. The basic idea in their work is to prepare the software for change in order to minimize efforts required for maintenance and especially when an architecture for a family of products is designed. They study some sources of variations that are composed of: *Variation in function*, *Variation in data*, *Variation in control flow*, *Variation in technology*, *Variation in quality goals* and *Variation in environment*. They classify the variation in software architecture regardless of the source of variation into: *a variation can be optional*, *a variation can be an instance out of several alternatives (XOR-Group)* and *a variation can be a set of instances out of several alternatives (OR-Group)*. They propose two basic techniques implement variation:*module replacement* and *data controlled variation*. The former is the technique of having multiple code-based versions of a particular module and choosing the correct one. The latter is the technique of maintaining the variation information in a data structure and having a single module that understands how to navigate the data structure to determine the correct actions.

In [Matinlassi, 2004], Matinlassi compares five methods for product line architectural design: *COPA, FAST, FORM, KobrA* and *QADA*. Most of these methods do not focus on SPLA as software development process because they define a full product line engineering process with activities and artifacts i.e., architecture, process, business, organization and etc.

Existing approaches that are interested to extract FM from available software artifacts of product variants is related to our work because the main source code of variability in SPLA is the variability at features level. These approaches as a follows:

In [Ryssel *et al.*, 2011], Ryssel et al. use formal concept analysis to generate a feature model. The input of their approach is a set of product configurations. The process of extraction feature model is based on NP-hard problem (e.g., set cover to identify OR-groups). Furthermore, architecture variability is not taken into account in this approach.

In [Acher *et al.*, 2011], Acher et al. propose an approach to reverse engineering architectural feature model. Their approach is based on the software architect's knowledge, architecture plugins dependencies and feature model extracted based on a reverse engineering approach proposed by She et al. [Steven *et al.*, 2011]. The basic idea in the proposed approach is to take the software architect's variability point of view to extract feature model so their work is named architectural feature model. However, the major limitations of this approach are firstly that the software architect is not available in most cases of legacy software products, and secondly that the architecture plugins dependencies are generally missing, too.

In [Steven *et al.*, 2011], She et al. propose an approach to extract feature model of given set of features. The input of the extraction process is feature names, feature descriptions and dependencies among available features. Based on this input data, they recover feature groups and cross tree constrains (i.e., *Require* and *Exclude*). A strong assumption behind this approach is that feature names, feature descriptions and dependencies among features are available. However, feature dependences are not always available especially in legacy software product variants.

4.2.3 Evaluation of Building SPLA Approaches

4.2.3.1 Evaluation Criteria

We consider a set of criteria relevant to our contribution to evaluate the studied approaches. This contribution aims to support reverse engineering SPLA from product variants by documenting commonality and variability at the architectural level as an important step toward this reverse engineering task.

- **Type of engineering process:** whether the studied work uses reverse engineering or forward engineering processes to build SPLA. Only in the reverse engineering process, we can exploit the existing software product variants to build SPLA.
- **Documenting commonality and variability at the architectural level:** whether the studied work defines mandatory components and VPs of components. This organization of components represents the variability aspects in SPLA, and hence enable to derive concrete architecture for each product in SPL.
- **Multiplicity of variability traceability between feature and architectural levels:** whether the studied work creates one-to-one mapping between variation points at feature level and variation points at architectural level. Such a mapping between variation points at both feature and architectural levels provides more flexibility and evolvability of SPLA [Pohl *et al.*, 2010].

4.2.3.2 Evaluation

Table 4.2 summarizes the studied approaches according to the evaluation criteria. As shown in this table, all studied approaches do not exploit existing product variants to reverse engineering SPLA. This is because these approaches represent forward engineering way to build SPLA, which means SPLA is developed from scratch. In forward engineering, the SPLA development is driven by the FM so that components are developed to implement the features offered by the FM.

Table 4.2 : Summary of approaches supporting SPLA development.

Studied Work	Type of engineering process	Documenting commonality and variability at the architectural level	Multiplicity of variability traceability between feature and architectural levels
[Sochos *et al.*, 2006]	Forward Engineering	no	——
[Trinidad *et al.*, 2007]	Forward Engineering	yes	one-to-one
[Zhang *et al.*, 2008]	Forward Engineering	yes	one-to-one
[Liu et Mei, 2003]	Forward Engineering	no	——
[Diana L. et Hassan, 2004]	Forward Engineering	yes	one-to-many
[Bachmann et Bass, 2001]	Forward Engineering	yes	——

The works of Trinidad et al. [Trinidad *et al.*, 2007] and Zhang et al. [Zhang *et al.*, 2008] documented commonality and variability at architectural level as mandatory components and a single group of optional components (i.e., a VP of optional components). This organization is inspired from the organization of features in the FM, as SPLA encompasses the commonality and variability in the FM. Moreover, these works established one-to-one mapping between VPs at feature level and VPs at architectural level as a single group of optional features is mapped to a single group of components. Also in these works, a component is created to each feature. The work of Diana and Hassan [Diana L. et Hassan, 2004] documented variability at architecture level as components equipped with VPs as we have seen. The source of variability at the architecture level is the variability at requirement level. In their model, a VP at requirement level may be mapped to one or more component, as each component is equipped one or more VPs. The remaining approaches do not pay attention to describe commonality and variability at the architectural level.

4.3 Conclusion

In this chapter, we have presented state-of-the-art information about feature-level CIA and development SPLA, as these subjects represent the applications of feature location in our work.

In the literature CIA approaches are organized into two categories: traceability-based CIA and dependency-based CIA. Feature-level CIA is traceability-based CIA. It is used to predict and determine affected features (resp. their implementations) for a given change proposal. Most existing approaches perform CIA at the source code level while few works perform CIA at other different levels of abstraction. The work proposed by Revelle et al. [Revelle *et al.*, 2011] is the only one that performs CIA at the feature level. This work does not provide quantitative measures for SPL's manger to help him for decision making. In their work, a feature (resp. its implementation) is described as affected or not affected (boolean value) without any indicator about to which degree the implementation of a feature is affected and also the percentage of affected features. In addition, they use a threshold mechanism to determine affected features. According to this mechanism, the implementations of coupled features under a given threshold are not investigated in spite of these features being affected by a change (as shown in their experimental results). For these limitations, we propose feature-level CIA which allows SPL's manger for decision making concerning change management. This approach meets all criteria mentioned in this chapter.

The SPLA is a key asset within SPL's core assets. It enables creation of architecture for each product in SPL. As such, it must support the scope of the product line and encompass commonality and variability among SPL members. Therefore, developing SPLA from scratch is a costly task. This has lead to the need for exploiting the existing product variants for creating SPLA in order to reduce the development cost. In the literature, all existing approaches support building SPLA from scratch (forward engineering for development) while reverse engineering SPLA from product variants has not been considered. Therefore, we propose an approach that takes advantage of the existence of product variants to support revere engineering SPLA from these variants, as we will have seen in chapter 7.

Part III

Contributions

CHAPTER 5

Feature Location in a Collection of Product Variants: Reducing IR Spaces

Preamble

In this chapter, we propose our approach to improve the IR-based feature location in a collection of object-oriented product variants. This improvement follows two strategies: reducing feature and source code spaces where IR applies and reducing abstraction gap between feature and source code levels. In section 5.1, we introduce these strategies. Next, sections 5.2 and 5.3 present core assumptions used in our approach and motivate the need for these strategies. In section 5.4, we give an overview about our feature location process. Sections 5.5 and 5.6, we present how our approach implements the strategies considered for improvement. In section 5.7, we describe the mapping between features and their implementing classes. In section 5.8, we show and explain the experimental evaluation of our approach and the obtained results. Finally, we conclude the chapter in section 5.9.

5.1 Introduction

The conventional application of IR for locating feature implementations in a collection of product variants involves conducting a textual matching between all features (i.e., their descriptions) and source code elements (e.g., classes, methods, etc.) of each product in product variants independently. On one hand, this may lead to retrieve irrelevant source code elements for features, especially in case of having similar features and trivial descriptions. On the other hand, as a feature implementation span multiple source code elements, this means that feature description is scattered across vocabulary used by these elements. Therefore, such conventional application hinders IR to find textual matching between features and their implementing source code elements. As a result, this leads to reduce the relevant source code elements that should be retrieved by IR.

In this chapter, we introduce our first contribution, which improves the conventional application of IR for locating feature implementations in a collection of object-oriented product variants. This improvement involves following two strategies.

Firstly, we exploit commonality and variability across product variants for reducing IR spaces (i.e., feature and source code spaces) into portions of features and their corresponding portions of source code elements. These portions represent minimal disjoint sets of features and their corresponding source code elements. These sets at the feature level represent the smallest number of features so that together their implementations can be isolated from source code of product variants. Moreover, these sets are disjoint (i.e., no shared features among them).

Secondly, we reduce the abstraction gap between the features and source code of the new obtained IR spaces by introducing the concept of *code-topic*. A *code-topic* is a cluster of similar classes that have common terms and they also are called near to each other. A *code-topic* can be a function implemented by the source code and provided by a feature. A *code-topic* allows the grouping of terms describing a feature implemented by code-topic's classes together rather than scattering these terms over many source code elements.

We mainly rely on *Formal Concept Analysis (FCA)* to reduce IR spaces. Also, we identify code-topics by investigating the results of two techniques: *Agglomerative Hierarchical Clustering (AHC)* and FCA. IR namely, LSI is used to link each feature to their implementing source code elements by taking the advantages of reducing feature and source code spaces, and reducing the abstraction gap between feature and source code levels.

5.2 Core Assumptions

The proposed IR-based feature location approach focuses on functional features. This is because typically software product variants implement a set of functional features [El Kharraz *et al.*, 2010]. Thus, we consider the following definition of the feature [Kang *et al.*, 1990]: "Feature is a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems". This definition focuses on external functional aspects of software system(s) that are observed by end users. These aspects represent functional features provided by product variants. Also, the proposed approach relies on the following assumptions:

- We assume that the functional feature is implemented by a set of classes. The concept of *class* represents a main building unit in all object-oriented languages and most often developers think about the class as a set of responsibilities that simulate a concept or functionality from the application domain [Marcus et Poshyvanyk, 2005].

- We assume that the functional feature always has the same source code classes where it is implemented. This is because product variants are developed by ad-hoc copying of existing feature implementations of existing variants. Therefore, the feature that is provided by different variants may have the same implementation.

- We assume that developers use the same vocabularies to name source code identifiers across product variants. This is because any product in product variants is developed by ad-hoc copying of existing products, which means there is a common vocabulary between identifiers across product variants.

5.3 Motivating the Reduction of IR Search Spaces

Our focus here is in applying IR-feature location techniques in the context of product variants where a feature can be implemented by multiple variants. As mentioned earlier, these variants provide features (resp. their source code elements) that are shared among all variants (mandatory features), among some variants and product-specific features. By considering software product variants together as a set of similar and related products, we can get additional input to IR-based feature location process. This input represents commonality and variability distribution across product variants. Commonality refers to mandatory features (resp. their source code elements) that are part of each product in product variants. Variability refers to optional features (resp. their source code elements) that are part of one product or more (but not all product variants). Analyzing commonality and variability across product variants allows the reduction of IR spaces into minimal disjoint set of features. This reduction enables a textual matching between the smallest number of features (i.e., their descriptions) and the source code elements that only implement those features. This leads to reducing the number of irrelevant source code elements (false-positive links) retrieved by IR techniques.

By grouping source code classes that collaborate together to constitute a code-topic, the number of retrieved source code classes that are relevant to a given feature may increase. To clarify the idea behind the *code-topic*, we exemplify it on the following example. Consider that a document Q contains the definition of the SPL and also assume that there are other documents so that each contains a part of this definition according to three scenarios. The first scenario consists of ten documents; the second scenario consists of four documents while the last one consists of two documents. The documents of each scenario together constitute the entire SPL's definition. Then, IR is used to compute the textual similarity between the document Q and the other documents according to the mentioned scenarios. The obtained textual similarity in the first scenario as [min, max] is [17%, 69%] while the similarity in the second and third scenarios respectively is [61%, 78%] and [86%, 88%]. Based on the obtained similarity in each scenario, we notice that the similarity incrementally increases along with grouping together documents so that the documents of the third scenario are more relevant to the Q. This because the these documents have more information related to Q. The concept of code-topic is based on this idea. Therefore, we propose the *code-topic*, as a coherent cluster of classes that have

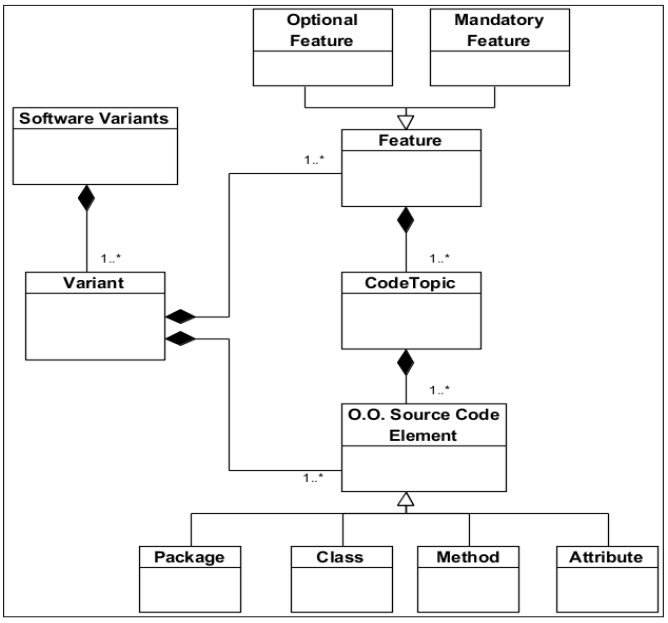

Figure 5.1 : Representation of the Relationship between Feature, Code-Topic and Object-Oriented Source Code Elements by MetaModel.

common terms and can be called near to each other. Figure 5.1 shows a meta model depicting the relationship between feature, code-topic and object-oriented source code elements.

The concept of code-topic also represents another aspect of reducing IR source code space. This aspect is concerned with reducing the number of established links between features and source code elements (classes). Before introducing the code-topic, the links are established between features and source code elements independently with mapping many-to-many while, after introducing the concept of code-topic, the links are established between features and code-topics (i.e. cluster of source code elements). Figure 5.2 clarifies this reduction. Such a reduction decreases the opportunity of establishing false-positive links between features and individual source code elements, as having many source code elements leads to a high probability to establish false-positive links. On the other hand, such a reduction increases the opportunity of linking together a set of source code elements (code-topic) that collaborate supporting a function to the same feature. As a result, such a reduction allows improvement to the accuracy of feature location.

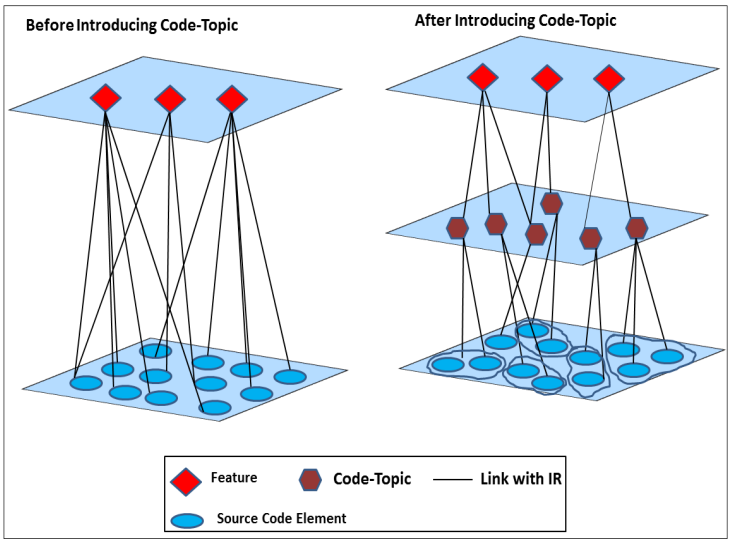

Figure 5.2 : Established Traceability Links before and after Introducing Code-Topics.

5.4 Feature Location Process in Our Approach

To implement the strategies mentioned in the previous section, we propose the process shown in Figure 5.3. This figure shows that our feature location process consists of three phases: *reducing IR (namely, LSI) spaces, reducing the abstraction gap between feature and source code levels* and *identifying feature-to-code's classes traceability links.*

In the first phase, we group all features (resp. their classes) of a given set of product variants into common and variable partitions at feature and source code levels. At the feature level, common and variable partitions respectively consist of all mandatory and optional features across product variants. At the source code level, common and variable partitions respectively consist of source code classes that implement mandatory and optional features. Also in this phase, the variable partitions at both levels are further fragmented into minimal disjoint sets. Each minimal disjoint set at feature level corresponds to its minimal disjoint set at source code level. In the second phase, we identify *code-topics* from the partition of common classes and each minimal disjoint set of classes. In the third phase, we link features and their possible corresponding *code-topics* using LSI. Such linking is used as a means to connect features with their source code classes by decomposing each code-topic to its classes. In the coming sections, we detail these phases.

As an illustrative example through this chapter, we consider four variants of a bank system, as it is shown in the Table 5.1, *Bank_V1.0* supports just core features for any bank system: *CreateAccount, Deposit, Withdraw* and *Loan. Bank_V1.1* has, in addition to the core features, *OnlineBank, Transfer* and *MobileBank* features. *Bank_V1.2* supports not only core features but also new features: *OnlineBank, Conversion, Consortium* and *BillPayment. Bank_V2.0* is an advanced application. It supports all previous features together.

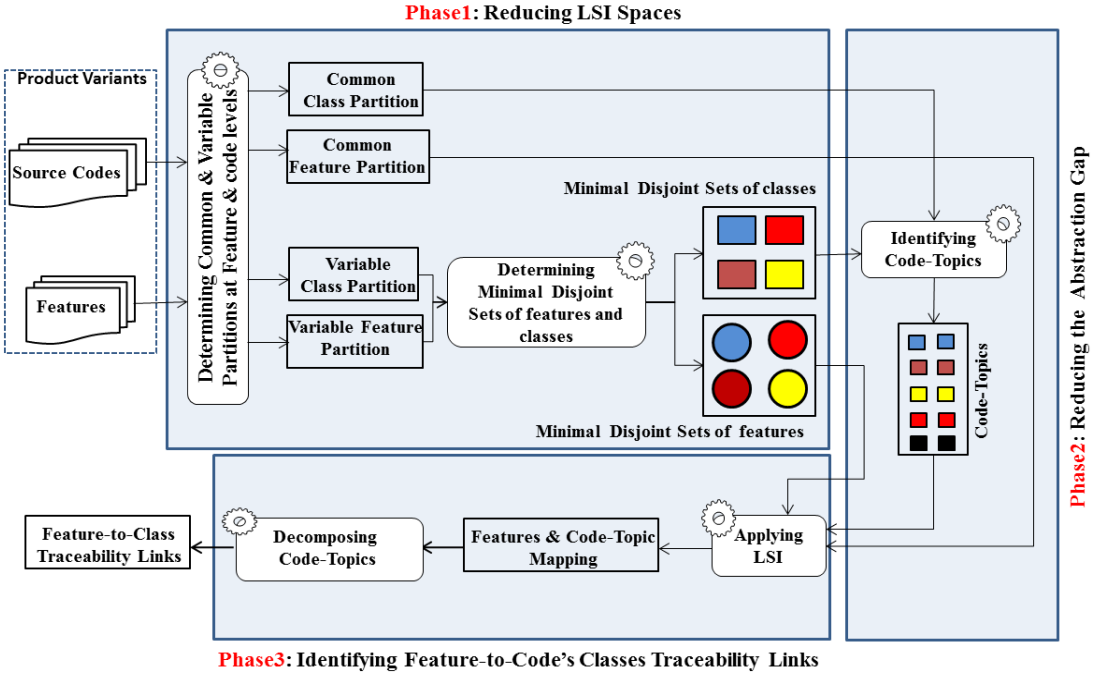

Figure 5.3 : An Overview of our Feature Location Process.

Table 5.1 : Feature set of four text bank systems.

Variant	Features
Bank_V1.0	Core (CreateAccount, Deposit, Withdraw, Loan).
Bank_V1.1	Core, OnlineBank, Transfer, MobileBank.
Bank_V1.2	Core, OnlineBank, Conversion, Consortium, BillPayment.
Bank_V2.0	Core, OnlineBank, Transfer, Conversion, Consortium, BillPayment, MobileBank.

5.5 Reducing IR Search Spaces

In this section, we address how to improve the conventional application of IR techniques (namely, LSI) for locating features in a collection of product variants by reducing feature and source code spaces. For this, we need to analyze and understand the commonality and variability distribution across product variants. Such analysis allows the reduction of IR spaces into fragments. We follow two steps at feature and source code levels for performing the reduction. The first step at both levels aims to determine common and variable partitions of features and source code's classes by analyzing commonality across product variants. The second step at both levels aims to fragment variable partitions of features and classes into minimal disjoint sets by analyzing variability using FCA. Figure 5.4 highlights how commonality and variability analysis can be used to reduce feature and source code spaces. In this figure, green portions refer to commonality while other colored portions refer to vari-

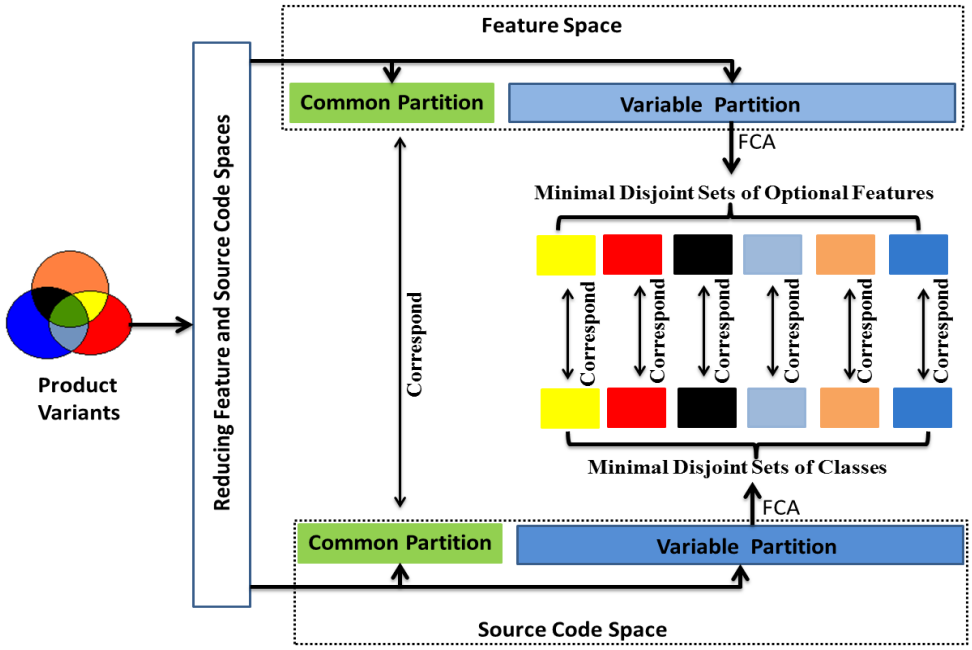

Figure 5.4 : Commonality and Variability Analysis across Product Variants.

ability distribution across product variants. Each colored portion at the feature level is composed of one or more features while the colored portion at the source code level is composed of source code classes that implement the corresponding portion of features. Below, we detail these steps.

5.5.1 Determining Common and Variable Partitions at the Feature Level

Product variants may have features with the same descriptions but have different names, due to changes in software environment or to the adoption of different technology [Yinxing *et al.*, 2010]. Therefore, we should determine correspondences among features across all product variants before going to reduce the feature space. We rely on Longest Common Subsequence (LCS) to find these correspondences [Bergroth *et al.*, 2000]. LCS computes pair-wisely the longest common subsequence of terms for two feature descriptions. We consider two features identical if and only if they have the same subsequence terms of their descriptions. Then, we rename corresponding features with the last name used.

As the common partition at the feature level is composed of mandatory features of product variants, we do textual matching among all feature names of product variants to find these mandatory features. The features that are part of each product in product variants form the common partition while the remaining features (i.e., optional features) in each variant form together the variable partition. In our illustrative example, all core features form the common partition while *OnlineBank, Transfer, Conversion, Consortium, BillPayment and MobileBank* features form the variable partition.

5.5.2 Determining Common and Variable Partitions at the Source Code Level

To group the source code classes of a given collection of product variants into common and variable partitions, we need to compare classes of product variants. Such comparison helps to determine classes that are part of each product (common partition of classes), and hence the remaining classes form the variable partition. For this, we first represent the source code classes for each variant as a set of Elementary Construction Units (ECUs). Each ECU has the following format:

$$ECU = PackageName@ClassName$$

This representation is necessary to be able to compare source code classes of product variants. Each product variant P_i is abstracted as a set of *ECUs* as follows: $P_i = \{ECU_1, ECU_2, \ldots, ECU_n\}$. In this way, an *ECU* reveals differences in source code's packages and classes from one variant to another variant (e.g. adding or removing packages or classes). These differences at the source code level occur due to adding or removing features to create a new variant. Common *ECUs* shared by all product variants represent common classes that form the common partition at source code level. The remaining *ECUs* in each variant represent classes that form together the variable partition at source code level. The common *ECUs* are computed by conducting a textual matching among *ECUs* of all product variants, as we assume that developers use the same vocabulary to name source code identifiers. The representation of source code as ECUs is inspired from [Blanc *et al.*, 2008].

5.5.3 Fragmentation of the Variable Partitions into Minimal Disjoint Sets

In this step, we reduce further the variable partitions at the feature and source code levels computed in the previous step. Our approach aims to fragment these partitions into minimal disjoint sets of optional features and their respective minimal disjoint sets of classes by using FCA (see Figure 5.4). These sets represent the final output of the process of reducing IR spaces. These sets are minimal because each set can not be reduced more. They also are disjointed because there are no shared members among them.

5.5.3.1 Determining Minimal disjoint Sets of Optional Features with FCA

Minimal disjoint sets of optional features can be obtained by analyzing optional features distribution across product variants. This analysis involves comparing all optional features of product variants. With referring again to Figure 5.4, we can see an example of optional features distribution across product variants. *Blue*, *red* and *orange* portions in this figure refer to features (resp. their classes) that are specific to *Product1*, *Product2* and *Product3* respectively. Black, yellow and grey portions refer to features that are shared between products pair-wisely. Such analysis allows determining minimal disjoint of features. We rely on FCA to perform such analysis in order to group optional features via concepts of GSH into minimal disjoint sets.

We apply FCA to all variant-differences extracted from a collection of product variants. For two product variants P_1 and P_2, we create three variant-differences. $P_1 - P_2$ (a set that contains features existing only in P_1 but not in P_2), $P_2 - P_1$ (a set that contains features existing only in P_2 but not in P_1) and $P_1 \bigcap P_2$ (a set that contains all the optional features that P_1 and P_2 have in common). If we consider, for example, variants Bank_V1.2 (V1.2) and Bank_V2.0 (V2.0) of our illustrative example, three variant-differences can be created as follows: V1.2-V2.0 = ϕ , V2.0-V1.2 = {*Transfer, MobileBank*}

Table 5.2 : Formal context for describing variant-differences of bank systems.

	OnlineBank	Transfer	Consortium	BillPayment	Conversion	MobileBank
$V1.2 - V1.0$	X		X	X	X	
$V1.0 - V2.0$						
$V2.0 - V1.0$	X	X	X	X	X	X
$V1.1 - V1.2$		X				X
$V1.2 \cap V2.0$	X		X	X	X	
...						

and V1.1 $\bigcap$ V1.2 = {*OnlineBank, Conversion, Consortium, BillPayment*}. The variant-differences aim at identifying all possible common and differences between each pair of variants in terms of provided features taking into account all combinations between product variants pair-wisely. Such strategy for using FCA to reduce IR search spaces is similar to the one proposed by Xue et al. [Xue *et al.*, 2012].

After extracting all variant-differences of a given collection of product variants, we define the formal context of FCA as follows:

- Variant-differences represent objects (extent)
- Optional features represent attributes (intent)
- The relation between an object and attribute refers to which optional feature is possessed by which variant-difference.

Table 5.2 shows the formal context of optional features and variant-differences corresponding to our illustrative example. Figure 5.5 shows the resulting GSH corresponding to the formal context defined in Table 5.2. Such a GSH shows the distribution of optional features across product variants. Each concept in this GSH consists of three fields. The upper field refers to a concept name (generated automatically). The middle field represents a set of optional features (intent). The bottom field shows variant-differences (extent). We are interested in the concepts associated with a set of optional features (such as the *Concept_5* in Figure 5.5) because these features represent the minimal disjoint set of features. The variant-differences of these concepts determine which product variants should be compared to obtain the implementation of these features.

5.5.3.2 Determining Minimal Disjoint Sets of Classes with FCA

For a given concept having a minimal disjoint set of features as intent, the extent (i.e. variant-differences) of such a concept determine which product variants should be compared to determine

classes that implement only those features in that concept. The obtained classes represent the minimal disjoint set of classes corresponding to the minimal disjoint set of features in that concept. In this way, we encounter two cases. Firstly, if the concept's extent is not empty, we randomly select only one variant-difference from the variant-differences listed in its extent. For instance, to compute the minimal disjoint set of classes corresponding to the minimal disjoint set of features for $Concept\_5$ in Figure 5.5, we can select the first variant-difference $(V2.0 - V1.1)$ to identify a set of classes that are present in $Bank\_V2.0$ but absent in $Bank\_V1.1$. The resulting set of classes implements only the features located in the $Concept\_5$ (*Consortium, BillPayment* and *Conversion*). Secondly, if the concept's extent is empty. i.e. it does not have own variant-differences but it inherits in a down-up manner variant differences from other concepts. In this case, we randomly select only one variant-difference from each concept located immediately below and directly related to this concept. For example, to compute the minimal disjoint set of classes corresponding to the minimal disjoint set of features for $Concept\_6$ in Figure 5.5, we select randomly a variant-difference from $Concept\_1$ and another one from $Concept\_2$. Considering that the selected variant-differences are (V1.1 $\bigcap$ V2.0 and V1.2 $\bigcap$ V2.0). According to these variant-differences, classes that implement the $OnlineBank$ feature located in $Concept\_6$ are present at the same time in $Bank\_V1.1$, $Bank\_V2.0$ and $Bank\_V1.2$.

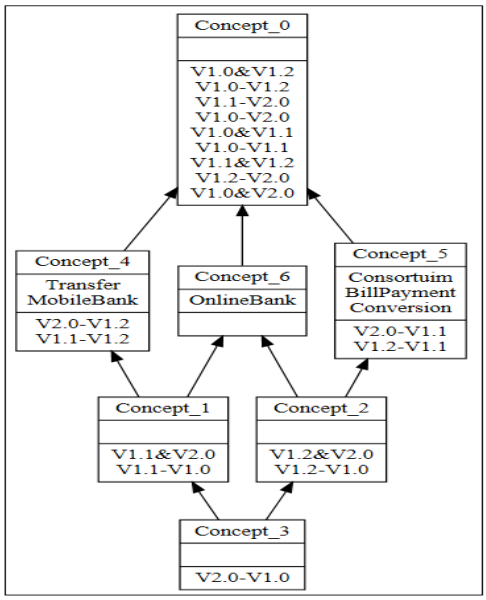

Figure 5.5 : GSH for the Formal Context of Table 5.2.

In both cases, commonalities and differences among source code classes of product variants are computed by lexically comparing their ECUs. For example, the corresponding classes of features of $Concept\_5$ are a set of *ECUs* that are present in $V1.2$ but are not present in $V1.1$.

5.6 Reducing the Abstraction Gap between Feature and Source Code Levels Using Code-Topic

As features and source code belong to different levels of abstraction, we propose code-topic to reduce the gap between these two levels of abstraction. In the section 5.3, we motivated the concept of code-topic. In this section, we address how to reduce this gap using code-topic in a collection of product variants. For this, we propose to identify code-topics from the common partition's classes and any minimal disjoint set of classes in order to benefit from the reduction of IR spaces. As the code-topic is a cluster of classes, we first formulate the identification of the code-topic as a partitioning problem. Next, we compute similarity among a given set of classes, as the code-topic is a cluster of similar classes. This similarity refers to textual similarity and structural dependency. Textual similarity refers to textual matching between terms derived from identifiers of the source code's classes. Structural dependency refers dependencies among classes (e.g. method call, inheritance, etc.) where classes that depend on each other are expected to constitute the members of the same code-topic. Then, we cluster similar classes together as code-topics by using a clustering technique.

Figure 5.6 shows an example to exemplify the code-topics identification process. In this figure, we assume that a given collection of product variants provides a set of features {*F1, F2, ..., F13*} and contains a set of classes {*C1, C2, ..., C47*}. This figure shows that similarity among the common partition's classes and each minimal disjoint set of classes is separately computed. Next, similar classes are clustered together to identify code-topics. Then, LSI is used to link the identified code-topics to their features.

5.6.1 Code-Topic Identification as a Partitioning Problem

According to our definition of the *code-topic*, the content of the *code-topic* matches a set of classes. Therefore, in order to determine a set of classes that can represent a *code-topic*, it is important to formulate the code-topic identification as a partitioning problem. This is because we need to group the classes of the common partition and any minimal disjoint set into groups. Each resulting group (partition) can be a candidate code-topic. The input of the partition process is a set of classes (C). This set could be classes of the common partition or classes of any minimal disjoint set (see Figure 5.6). The output is a set of *code-topics* (T) of C. $C = \{c_1, c_2, ..., c_n\}$ and $T(C) = \{T_1, T_2, \ldots, T_k\}$ where:

- c_i is a class belonging to C.
- T_i is a subset of C.
- k is the number of identified *code-topics*.
- $T(C)$ does not contain empty elements: $\forall T_i \in T(C), T_i \neq \Phi$.
- The union of all $T(C)$ elements is equal to C: $\bigcup_{i=1}^{k} T_i = C$. This property is called completeness.

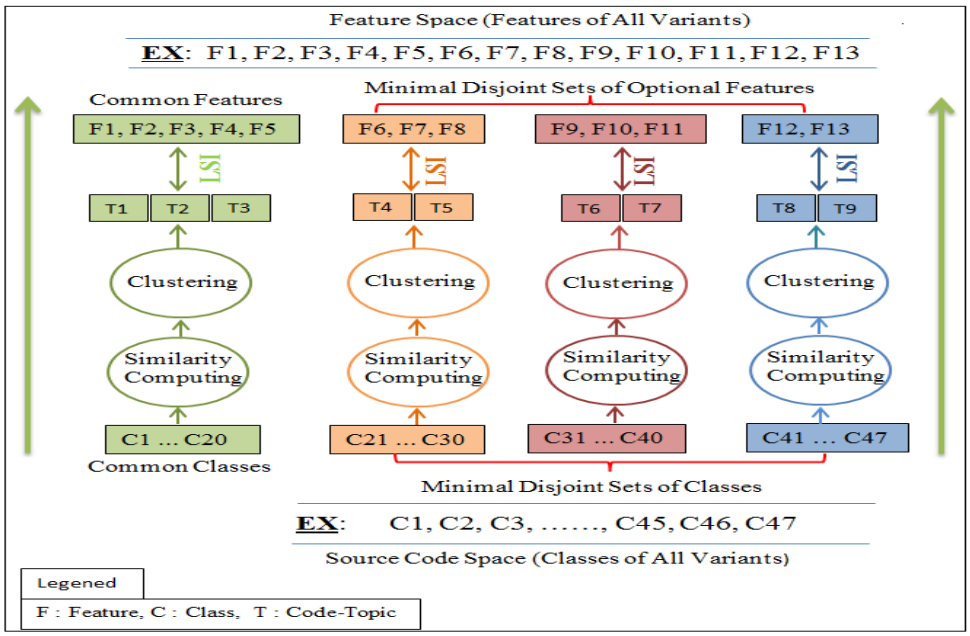

Figure 5.6 : An Overview of the Code-Topic Identification Process.

5.6.2 Computing Similarity Between Classes for Supporting Code-Topic Identification

As the code-topic is a cluster of similar classes, we need to compute this similarity which refers to textual similarity and structural dependency between classes to support the process of code-topic identification. In the following, we detail these two types of similarity.

5.6.2.1 Textual Similarity

To compute textual similarity among given sets of classes, we rely on VSM. We follow the same steps of VSM explained in section 2.2 of preliminaries chapter however we create a document for each class. Each document is a list of all identifiers of its corresponding class. All class documents are used as queries and corpus documents at the same time for VSM. In VSM, the textual similarity between two class documents is measured by using cosine similarity between their corresponding vectors. One of these documents is treated as a query while the other is treated a corpus document. Two documents are considered similar if the cosine of the angle between their corresponding vectors is greater than or equal to 0.70. As we mentioned in the preliminaries chapter, this value represents the most widely used threshold for the cosine similarity [Marcus et Maletic, 2003b]. After computing the cosine similarity among all class documents, we build a cosine similarity matrix whose columns and rows are identical and represent the class documents. An entry in this matrix refers to the cosine similarity value. As an example of a matrix, we can imagine the cosine similarity matrix of classes of a bank system using the illustrative example as shown in Table 5.3.

Table 5.3 : The cosine similarity matrix of the illustrative Example.

	Bill@BillAcount	**Conversion@converter**	**Bill@OldBills**	**Transfer@TargetAccount**	**Bill@PayPartially**	**Conversion@CurrencyInfo**	**Bill@PaymentMethod**	**Transfer@SourceAccount**	...
Bill@BillAcount	100%	0.0%	75%	0.0%	75%	0.0%	75%	0.0%	
Conversion@converter	0.0%	100%	0.0%	0.0%	0.0%	75%	75%	0.0%	
Bill@OldBills	75%	0.0%	100%	0.0%	75%	0.0%	75%	0.0%	
Transfer@TargetAccount	0.0%	0.0%	0.0%	100%	0.0%	0.0%	0.0%	75%	
Bill@PayPartially	75%	0.0%	75%	0.0%	100%	0.0%	75%	0.0%	
Conversion@CurrencyInfo	0.0%	75%	0.0%	0.0%	0.0%	100%	75%	0.0%	
Bill@PaymentMethod	75%	75%	75%	0.0%	75%	75%	100%	0.0%	
Transfer@SourceAccount	0.0%	0.0%	0.0%	75%	0.0%	0.0%	0.0%	100%	
...									

5.6.2.2 Structural Dependency

To compute structural dependency among a given set of classes, we should determine classes that depend on each other. Therefore, we rely on a coupling metric that captures different types of interactions among classes. These interactions include [Shyam et Chris, 1994] [Churcher et Shepperd, 1995][Sun *et al.*, 2011]:

1. **Inheritance relationship:** when a class inherits attributes and methods of another class.
2. **Method call:** when method(s) of one class use method(s) of another class.
3. **Attribute access:** when method(s) of one class use attribute(s) of another class.
4. **Shared method invocation:** when two methods of two different classes invoke the same method belonging to a third class.
5. **Shared attribute access:** when two methods of two different classes access an attribute belonging to a third class.

These interactions among classes allow us to determine a cohesive unit of classes as code-topic should be. The structural dependency is computed pair-wisely between classes. We consider two classes to depend on each other if they have at least one of the above mentioned interactions. We quantify structural dependency as discrete values either `1` (if there is an interaction between the two focused classes) or `0` (if there is no interaction). After computing the structural dependency among all classes, we build a structural dependency matrix. The structure of this matrix is the same as the cosine similarity matrix shown in Table 5.3. An entry in the structural dependency matrix refers to the value of structural dependency.

5.6.3 Clustering Classes as Code-Topics

After computing the similarity between classes of a given set, in this section we cluster similar classes together as a candidate code-topic. Clustering techniques group similar classes together and aggregate them into clusters [Jain *et al.*, 1999]. Each cluster is considered as a code-topic. In our approach, we rely on two techniques for clustering purpose: Formal Concept Analysis (FCA) and Agglomerative Hierarchal clustering (AHC).

5.6.3.1 FCA for Code-Topics Identification

As we mentioned in the preliminaries chapter, FCA identifies meaningful groups of objects sharing common attributes. In our approach, we exploit this grouping property for FCA to identify meaningful groups (clusters) of similar classes. These groups represent code-topics. As we need to extract code-topics from each minimal disjoint set of classes and from the common partition of classes, we should create a separated formal context for each one. The objects and attributes for each formal context are identical and represent the same set of focused classes. The relation between objects and attributes refer to the similarity between classes. This similarity can be textual similarity, structural dependency or a combination thereof.

In case of using the textual similarity, the cosine similarity matrix (defined in the previous subsection) is used to build the formal context. As the values of this matrix are continuous while the formal context is a binary context (i.e. their values either 0 or 1), we can generate several binary contexts from this matrix by specifying threshold values $\beta \in [0,1]$. Therefore, we consider the most commonly used threshold for the cosine similarity ($\beta = 0.70$). All matrix values that are greater than or equal to β are scaled to `1` while other values are scaled to `0`. The obtained matrix after normalization represents the required formal context. In our illustrative example, the binary context that is extracted from Table 5.3 at $\beta = 0.70$ is shown in Table 5.4. In this table the symbol (*X*) refers to *1*.

In case of using the structural dependency, the structural dependency matrix (defined in the previous subsection) is used to build the formal context. The values of this matrix do not need a normalization because they are binary values by nature.

In case of using a combination of textual similarity and structural dependency, we consider two classes similar in the formal context if they at least are textually similar or depend on each other.

After building the required formal contexts, the concept GSH corresponding to each formal context is generated. In the resulting GSH, clusters of similar classes can be identified by the concepts having equal extent and intent sets. These concepts are known as square concepts [Azmeh *et al.*, 2011]. The extent of each square concept is a cluster of similar classes that can be a candidate *code-topic*. Each class in such a cluster is similar to all other cluster classes. Figure 5.7 shows the generated GSH corresponding to the formal context of our illustrative example defined in Table 5.4. The notion of square concepts can be better recognized by performing column-line interchange for the formal context defined in Table 5.4. The resulting interchanged context is shown in Table 5.5. The highlighted portions in this table represent clusters of similar classes (square concepts). From the interchanged context or from the GSH in Figure 5.7, we can identify the following code-topics:

- **Code-topic 1**: {*Bill_BillAcount, Bill_OldBills, Bill_PayPartially*}.

Table 5.4 : The formal context extracted from Table 5.3 at $\beta = 0.70$.

	Bill_BillAcount	Conversion_converter	Bill_OldBills	Transfer_TargetAccount	Bill_PayPartially	Conversion_CurrencyInfo	Bill_PaymentMethod	Transfer_SourceAccount	...
Bill_BillAcount	X		X		X		X		
Conversion_converter		X				X	X		
Bill_OldBills	X		X		X		X		
Transfer_TargetAccount				X				X	
Bill_PayPartially	X		X		X		X		
Conversion_CurrencyInfo		X				X	X		
Bill_PaymentMethod	X	X	X		X	X	X		
Transfer_SourceAccount				X				X	
...									

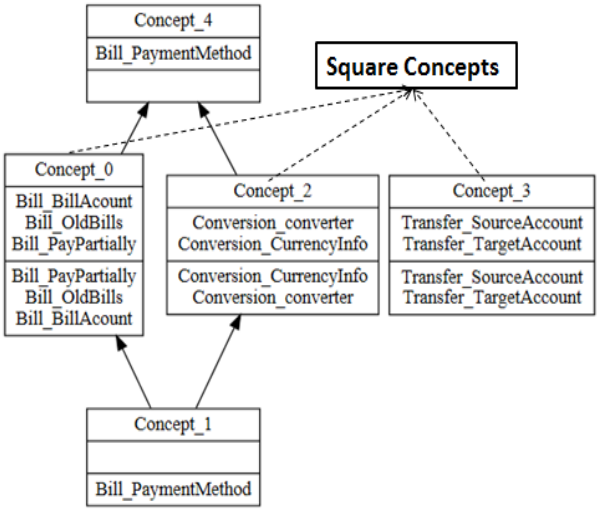

Figure 5.7 : The Corresponding GSH for the Formal Context Shown in Table 5.4.

- **Code-topic 2**: {*Conversion_converter, Conversion_CurrencyInfo*}.
- **Code-topic 3**: {*Transfer_SourceAccount, Transfer_TargetAccount*}.

Additionally, we considered the extent of non-square concepts (such as, *Concept_1* in Figure 5.7) as a *code-topic* because such a concept always inherits from its ascendants an intent set containing its extent members.

Table 5.5 : The interchanged formal context.

	Bill_BillAcount	Bill_OldBills	Bill_PayPartially	Bill_PaymentMethod	Conversion_converter	Conversion_CurrencyInfo	Transfer_SourceAccount	Transfer_TargetAccount	...
Bill_BillAcount	X	X	X	X					
Bill_OldBills	X	X	X	X					
Bill_PayPartially	X	X	X	X					
Bill_PaymentMethod	X	X	X	X	X	X			
Conversion_converter				X	X	X			
Conversion_CurrencyInfo				X	X	X			
Transfer_SourceAccount							X	X	
Transfer_TargetAccount							X	X	
...									

Although FCA can identify a cluster of classes such that each class must be similar to all other cluster's classes, it does not merge together similar clusters. For example, although *Bill_PaymentMethod* class is similar to classes Bill_BillAccount, Bill_OldBills and Bill_PayPartially, it appears in a separated concept as shown in Figure 5.7. Moreover, FCA deals with only binary context and needs a threshold mechanism to deal with multi-value context. In our case, this means that all cosine similarity values over (0.70) are equivalent, however these values are not. For example, considering three class documents (*D1*, *D2* and *D3*) and the cosine similarities between *D1* and other documents (*D2* and *D3*) are respectively (0.71 and 100). Based on FCA, *D1* is similar to *D2* and D3 with the same degree of similarity, as both values are scaled to 1. In the following, we show how AHC overcomes these limitations.

5.6.3.2 AHC For Code-Topics Identification

Agglomerative Hierarchal Clustering (AHC) is the second option to cluster similar classes into code-topics in our approach. AHC starts with singleton clusters (i.e. clusters having only one object) and recursively merges the two most similar cluster in each stage. In our approach, these singleton clusters initially consist of individual class documents and later of clusters of class documents formed during the previous stages. Based on this description of AHC, we can deduce that AHC computes similarity among class documents, among clusters, and between clusters and class documents. Therefore, it overcomes the limitation of FCA concerned with computing similarity only between class documents. Our application of AHC relies on the following two steps:

Building a Hierarchy of Clusters

For a given set of classes, AHC aggregate similar classes into clusters. The basis for clustering classes is the strength of the relationship between them. This relationship may refer to textual similarity, structural dependency or a combination thereof. In AHC, we use the textual similarity. We rely on VSM to compute this similarity like FCA.

Algorithm 1: BuildingDendrogram

Input: $classes$
Output: $DendrogramTree(dendgr)$

```
1  stack clusters ← classes
2  while (|clusters| > 1) do
3  |   (Clu1, Clu2) ← mostSimilarClusters(clusters)
4  |   Pop(Clu1, clusters)
5  |   Pop(Clu2, clusters)
6  |   Clu3 ← Merge(Clu1, Clu2)
7  |   Push(Clu3, clusters)
8  end
9  dendgr ← get(clusters)
10 return dendgr;
```

AHC works by creating a tree of nested clusters, called a dendrogram. A dendrogram is a tree diagram frequently used to illustrate the arrangement of the clusters produced by hierarchical clustering [Haifeng et Zijie, 2010]. We adapt AHC to build a dendrogram from a given set of class documents according to Algorithm 1. This algorithm relies on a series of successive binary mergers, initially of individual class documents and later of clusters formed during the previous stages. In the beginning, it puts each class document in its own cluster. Among all current clusters, the two most textually similar clusters (mostSimilarClusters()) are picked. Then, these two clusters are replaced with a new cluster by merging the two original ones. The process continues until only one cluster remains such that at each iteration only one pair of clusters that have the highest relationship strengths are merged. We obtain from this single cluster a dendrogram (dendgr) that contains a set of nested clusters. Based on this description about how clusters are formed, AHC overcome the limitation of FCA concerning with threshold mechanism used to scale multi-value context.

Figure 5.8 shows an example of dendrogram tree. At the lowest level, each class document is in its own cluster. At the highest level, all classes belong to the same cluster. The internal nodes represent new clusters formed by merging the clusters that appear as their children in the tree.

Selection of Candidate Code-Topics

Breaking the generated dendrogram tree based on predefined criteria groups classes into clusters. Each resulting cluster can be a candidate *code-topic*. Therefore, we must select the appropriate breaking points to obtain *code-topics*. This selection is performed by an algorithm based on a depth-first search (refer to Algorithm 2) [Kebir *et al.*, 2012a]. This algorithm takes as input the dendrogarm tree and returns a set of clusters. We interpret these clusters as *code-topics*. This algorithm starts by comparing the textual similarity value (Sim()) of each node in the dendrogram (starting from the root) and its sons. If the similarity value of the focused node is less than the average of the similarity values of its two sons, then the algorithm continues to the next son nodes. Otherwise, the focused node is identified as a *code-topic*, added to the *code-topics* accumulator (T) and the algorithm computes the next node in the stack (*traversedClusters*). In this way, the most relevant *code-topics* will be identified

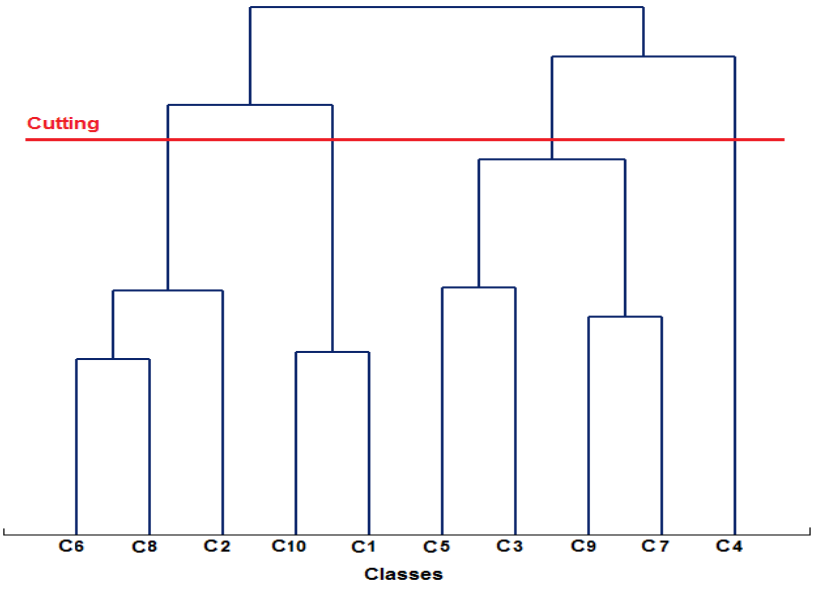

Figure 5.8 : An Example of Dendrogram Tree.

as the traversal continues.

To visualize how the algorithm 2 selects clusters (i.e.,*code-topic*), we again refer to Figure 5.8. The red horizontal line determines the cutting points. Based on these points, we obtain four clusters as follows. A first cluster contains only class `4`. A second cluster contains classes `5`, `3`, `9` and `7` while classes `10` and `1` belong to a third cluster. Finally, classes `6`, `8` and `2` form a fourth cluster. Each cluster represents a candidate *code-topic*.

5.7 Locating Features by LSI

In this section, we present how to link features with their implementing classes taking advantage of reducing IR search spaces and abstraction gap between feature and source code levels. We achieve this by following two steps: establishing a mapping between features and their respective code-topics and then decomposing the code-topics to their classes.

5.7.1 Establishing a Mapping between Features and Code-Topics

We rely on LSI to link features to their code-topics as shown in Figure 5.6. We link features of the common partition to the code-topics extracted from this partition. Also, we link features of each minimal disjoint set to code-topics extracted from its corresponding minimal disjoint set of classes.

LSI is applied by following the steps described in the preliminaries chapter (see section 2.2) however we build LSI's corpus and queries as follows. LSI's corpus consists of *code-topic* documents. We create a document for each code-topic containing terms extracted from identifiers of code-topic's classes. For LSI's queries, we create s document for each feature containing a feature name and the

Algorithm 2: CodeTopicDendrogramTraversal

Input: $Dendrogram(dendgr)$
Output: $Code\text{-}Topics(T)$

```
1  stack traversedClusters
2  push(root(dendgr), traversedClusters)
3  while (|traversedClusters| > 0) do
4      parent ← pop(traversedClusters)
5      son1 ← getSon1Cluster(parent, dendgr)
6      son2 ← getSon2Cluster(parent, dendgr)
7      avg ← average(Sim(son1), Sim(son2))
8      if (Sim(parent) > (avg)) then
9          add(parent, T)
10     else
11         push(son1, traversedClusters)
12         push(son2, traversedClusters)
13     end
14 end
15 return T
```

description. Each feature document represents a query. We rely on *Term frequency-inverse document frequency* (*tf-idf*) metric to assign a weight for each term extracted from feature and code-topic documents (refer to Equation 5.1). This metric is commonly used for the purpose of feature location [Ali *et al.*, 2011][Peng *et al.*, 2013].

$$W_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \times \log_2\left(\frac{|\,D\,|}{|\,d : t_i \in d\,|}\right) \tag{5.1}$$

In equation 5.1, $W_{i,j}$ is the weight the term of t_i in document d_j, $n_{i,j}$ is the number of occurrences of term t_i in document d_j, $\sum_k n_{k,j}$ is the sum of occurrences of all terms in document d_j, $|\,D\,|$ is the total number of documents in the collection, and $|\,d : t_i \in d\,|$ is the number of documents in which the term t_i appears.

LSI takes *code-topic* and feature documents as input. Then, LSI measures the similarity between the *code-topics* and features using the cosine similarity. LSI returns a list of *code-topics* ordered by their cosine similarity values against each feature. The *code-topics* retrieved should have a cosine similarity value greater than or equal to *0.70*, as this value represents the most widely used threshold for the cosine similarity [Marcus et Maletic, 2003b].

5.7.2 Decomposing Code-Topics into their Classes

After linking each feature to all its corresponding *code-topics*, we can easily determine relevant classes for each feature by decomposing each *code-topic* to its classes. For instance, imagine that we have a feature (F1) that is linked to two *code-topics*: *code-topic1= {c1, c4}* and *code-topic2= {c7, c6}*. By decomposing these code-topics into their classes; we find that *F1* is implemented by five classes *{c1, c4, c7, c6}*.

5.8 Experimental Evaluation

In this section, we provide a validation of our approach for supporting feature location in a collection of product variants. During this section, we present evaluation measures used. Also, we discuss the obtained results and present threats of validity to our approach.

5.8.1 Evaluation Measures

The effectiveness of IR techniques is commonly measured by their *precision*, *recall* and *F-measure* [Salton et McGill, 1986]. We adapt these metrics in our context as follows. Precision is the percentage of retrieved links (i.e. class documents) that are relevant to the total number of retrieved links. Recall is the percentage of retrieved links that are relevant to the total number of relevant links. 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. These measures can be computed for each feature separately, or for a set of features provided by a single product or a collection of product variants. Equations 5.2, 5.3 and 5.4 represent respectively *precision*, *recall* and *F-measure*.

$$Precision = \frac{|\{Relevant\ Links\} \bigcap \{Retrieved\ Links\}|}{|\{Retrieved\ Links\}|} \times 100\% \tag{5.2}$$

$$Recall = \frac{|\{Relevant\ Links\} \bigcap \{Retrieved\ Links\}|}{|\{Relevant\ Links\}|} \times 100\% \tag{5.3}$$

$$F\_measure = \frac{2}{\frac{1}{Reacll} + \frac{1}{Precision}} \times 100\% \tag{5.4}$$

All measures have values in a range [0, 1]. If the precision value equals 1, this means that all the retrieved links are relevant but also this does not mean that all relevant links are retrieved (i.e., there are false negative links). If the recall value equals 1, this means that all relevant links are retrieved but also this does not mean that all retrieved links are relevant (i.e., there are false positive links). Higher precision, recall and F-measure mean better results [Salton et McGill, 1986].

5.8.2 Results and Analysis

The most important parameter of LSI is the number of chosen term-topics. As mentioned in the preliminaries chapter (see section 2.2.3.2), this number differs from one case study to another and there is no recommended value for this parameter in the literature. In our work, we cannot use a fixed number of term-topics because we have different sizes of class documents and code-topic documents. Thus, we use a factor K between 0.1 and 0.5 as well as between 0.01 and 0.05 to determine the number of term-topics. The number of term-topics (*#Term-topics*) equals to "$k \times D_{dim}$", where D_{dim} is a document dimensionality of the term-by-document matrix that is generated by LSI.

We organize and explain the obtained results into three parts. In the beginning, we compare our approach (called feature-to-code tractability in product variants or FCT for short) with both the conventional application of LSI (Conv) and the most recent and relevant work on the subject, called

PL-PV (proposed by Xue et al. [Xue *et al.*, 2012]). Next, we compare results obtained using textual similarity, structural dependency and their combination with FCA to identify code-topics for supporting feature location. Finally, we compare the results obtained using FCA and AHC to identify code-topics for supporting feature location.

5.8.2.1 Comparing FCT, Conv and FL-PV Approaches

Table 5.6 summarizes the average precision, recall and F-measure values obtained by CONV and FCT approaches for all products considered of ArgoUML-SPL and MobileMedia case studies (section 2.4 details these case studies) at different values of K. We compute the values of these measures in both approaches as follows. We first determine the total number of retrieved links, the total number of retrieved links that are relevant and the total number of relevant links for all features of each product at each value of *K*. Next, we compute precision, recall and F-measure for each product at different values of K. Then, we compute the average of precision, recall and F-measure values for all products considered. The results of FCT shown in Table 5.6 is based on using textual similarity measure and FCA to identify code-topics. We can deduce from this table that recall and precision results of FCT are better than those of the CONV in both case studies. This is attributed to two main reasons. Firstly, FCT maps small sets of features to small sets of their respective source code classes by reducing the LSI spaces. Secondly, FCT bridges the abstraction gap between feature and source code levels using the *code-topics*. The F-measure results confirm that FCT gives higher precision and recall comparing with CONV. This is because when the F-measure is high, then precision and recall are also high (according to the F-measure definition).

Table 5.7 summarizes the precision, recall and F-measure values of FCT and FL-PV approaches at different values of *K*. We compute the values of these measures in both approaches as follows. We determine the total number of retrieved links, the total number of retrieved links that are relevant and the total number of relevant links for all features in a collection of variants of ArogUML-SPL and MobileMedia. Then, we compute precision, recall and F-measure for all features together of a given collection at different values of K. From the Table 5.7, we can deduce that FCT outperforms FL-PV in terms of precision, recall and F-measure metrics. This is attributed to the fact that FCT does not only consider reducing LSI spaces like FL-PV but also reduces the abstraction gap between feature and source code levels. This means that reducing LSI spaces is not the only important factor to improve LSI results but also reduction the abstraction gap is another important factor. In addition, Table 5.7 shows that FCT and FL-PV give the same results in case of MobileMedia for the following reasons. Firstly, most minimal disjoint sets of optional features consist of only one feature, and hence their corresponding minimal disjoint sets of classes contain only the implementation of that feature (no more and no less). Thus, in this case we do not require LSI and *code-topics*. Secondly, each of MobileMedia features is implemented by a few classes and sometimes by only two classes, and do not have enough information to build *code-topics*.

5.8.2.2 Comparing Textual Similarity, Structural Dependency and their Combination with FCA

Table 5.8 compares the precision, recall and F-measure values obtained by considering textual similarity, structural dependency and their combination as a similarity measure for identifying *code-topics*

Table 5.6 : Average Precision, Recall and F-measure of FCT and CONV.

Case Study	**ArgoUML-SPL**					
	Precision		**Recall**		**F-measure**	
K	**FCT**	**CONV**	**FCT**	**CONV**	**FCT**	**CONV**
0.01	51%	21%	99%	91%	68%	34%
0.02	52%	22%	86%	82%	65%	35%
0.03	52%	29%	85%	59%	65%	39%
0.04	52%	42%	87%	39%	65%	40%
0.05	56%	63%	73%	25%	63%	36%
Case Study	**MobileMedia**					
	Precision		**Recall**		**F-measure**	
K	**FCT**	**CONV**	**FCT**	**CONV**	**FCT**	**CONV**
0.1	70%	25%	100%	63%	82%	40%
0.2	70%	26%	99%	60%	82%	40%
0.3	79%	30%	84%	50%	82%	41%
0.4	79%	32%	80%	35%	80%	39%
0.5	84%	43%	76%	27%	80%	38%

using FCA. Also, Table 5.9 shows average precision, recall and F-measure values of similarity measures shown in the Table 5.8.

Based on these tables, we notice that only considering structural dependency achieves minor enhancement in precision, recall and F-measure values compared to considering only textual similarity, and their combination in case of ArgoUML-SPL. This minor enhancement is due to the fact that ArgoUML-SPL is a large-scale case study, which leads to a lot of interactions between its source code classes. Therefore, the structural dependency is a good choice to capture classes that depend on each other, which leads to identifying more relevant code-topics. This minor difference between the results of three measures confirm our assumption about code-topics' classes, as classes that contribute to form a code-topic sharing similar terms and are called near to each other. However, in case of MobileMedia, the three similarity measures give the same results. This is due to reasons already mentioned (see section 5.8.2.1).

5.8.2.3 Comparing AHC and FCA

Table 5.10 summarizes precision, recall and F-measure results obtained by using AHC and FCA for identifying code-topics for ArgoUML-SPL and MobileMdia.

On a large-scale system (ArgoUML-SPL), we notice that AHC significantly improves the recall val-

Table 5.7 : Precision, Recall and F-measure values of FCT against FL-PV.

Case Study	ArgoUML-SPL					
	Precision		Recall		F-measure	
K	FCT	FL-PV	FCT	FL-PV	FCT	FL-PV
0.1	70%	34%	40%	29%	51%	31%
0.2	57%	7%	9%	4%	16%	5%
0.3	57%	2%	5%	1%	9%	2%
0.4	62%	1%	4%	0%	8%	1%
0.5	57%	0%	2%	0%	3%	0%
Case Study	**MobileMedia**					
	Precision		**Recall**		**F-measure**	
K	**FCT**	**FL-PV**	**FCT**	**FL-PV**	**FCT**	**FL-PV**
0.1	85%	85%	100%	100%	92%	92%
0.2	85%	85%	100%	100%	92%	92%
0.3	93%	93%	93%	93%	93%	93%
0.4	93%	93%	93%	93%	93%	93%
0.5	96%	96%	89%	89%	93%	93%

Table 5.8 : Precision, recall and F-measure of textual similarity, structural dependency and their combination using FCA

Case Study	ArgoUML-SPL								
	Precision			Recall			F-measure		
K	Textual	Structural	Combination	Textual	Structural	Combination	Textual	Structural	Combination
0.1	70%	64%	71%	40%	51%	51%	51%	56%	59%
0.2	57%	95%	60%	9%	20%	6%	16%	33%	11%
0.3	57%	88%	89%	5%	3%	3%	9%	5%	6%
0.4	62%	67%	60%	4%	1%	2%	8%	2%	4%
0.5	57%	67%	80%	2%	0%	1%	3%	1%	2%
Case Study	**MobileMedia**								
	Precision			**Recall**			**F-measure**		
K	**Textual**	**Structural**	**Combination**	**Textual**	**Structural**	**Combination**	**Textual**	**Structural**	**Combination**
0.1	85%	85%	85%	100%	100%	100%	92%	92%	92%
0.2	85%	85%	85%	100%	100%	100%	92%	92%	92%
0.3	93%	93%	93%	93%	93%	93%	93%	93%	93%
0.4	93%	93%	93%	93%	93%	93%	93%	93%	93%
0.5	96%	96%	96%	89%	89%	89%	93%	93%	93%

Table 5.9 : Average precision, recall and F-measure of textual similarity, structural dependency and their combination using FCA

Case Study	Precision			Recall			F-measure		
	Textual	Structural	Combined	Textual	Structural	Combined	Textual	Structural	Combined
ArgoUML-SPL	61%	76%	72%	12%	15%	13%	17%	19%	16%
MobileMedia	90%	90%	90%	95%	95%	95%	92%	92%	92%

ues with a minor decrease in the precision compared to FCA. This improvement in recall is due to the fact that AHC identifies *code-topics* by determining a set of clusters so that classes of each cluster are similar among themselves and dissimilar to classes of other clusters. For FCA, it identifies *code-topics* by determining a set of clusters (i.e. square concepts) in which all cluster's members are similar to each other but it does not consider similarity between clusters. This means that FCA computes the textual similarity only among classes while AHC computes the similarity not only among classes but also among clusters. Therefore, the total number of *code-topics* extracted from all minimal disjoint set of classes using FCA is higher than AHC (423, 17 respectively). By identifying a small number of *code-topics*, we can get more relevant information describing features that are implemented by the identified *code-topics'* classes. Regarding the minor decrease in precision, this is due to the fact that AHC depends a lot on VSM compared to FCA. AHC uses VSM to compute similarity among classes, among clusters, and between classes and clusters while FCA uses VSM to compute similarity only between classes. This means that the number of false-positive links in the case of AHC is higher than FCA because VSM may retrieve false-positive links which leads to impression. Table 5.10 also shows that AHC significantly improves F-measure results compared to FCA. Considering the significant improvement in recall and minor decrease in precision. We believe that this improvement in F-measure is expected; as F-measure is a harmonic mean between recall and precision values. This means that using AHC leads to better compromising between precision and recall than FCA.

On a small system (MobileMedia), it is observed that AHC, FCA produce the same precision, recall and F-measure results because of the reasons already mentioned (see section 5.8.2.1).

Table 5.11 presents some code-topics extracted from the minimal disjoint partition of classes that correspond to two features of ArgoUML-SPL (*UseCase* and *Sequence* features). These code-topics are extracted based on AHC to cluster similar classes as code-topics. We present in this table a cluster of *ECUs* that correspond to each code-topic. We observe that classes of *code-topic 1* are related, as their names share the term (*ActionNew*) and most of them belong to the same package (*argouml.uml.ui.behavior.usecases*). Moreover, the term *UseCase* is shared among class and package names. Based on this observation, we can deduce that these classes collaborate to support a function provided by *UseCase* feature. This similarity in their class and package names indicate that the content of these classes is also textually similar, and hence AHC groups them as a code-topic. For *code-topic 2*, most the classes of this code-topic belong to packages that their names consists of the term *sequence* and also the names of these classes share the term *Fig*. Fig is acronym for ***Fig****ure*, as these classes are responsible for showing objects on a sequence diagram. Therefore, these classes may collaborate to support a function provided by *Sequence* feature. Also, we can notice that this code-topic contains two classes (*ActorPortFigRect* and *FigMyCircle*) belonging to a package (*argouml.uml.diagram.use_case.ui*) that may concern with *UseCase* feature. This is due to one of the

Table 5.10 : Precision, recall and F-measure of AHC against FCA for ArgoUML-SPL and MobileMedia.

ArgoUML-SPL						
	Precision		Recall		F-measure	
K	AHC	FCA	AHC	FCA	AHC	FCA
0.1	52%	70%	99%	40%	68%	51%
0.2	52%	57%	99%	9%	68%	16%
0.3	52%	57%	98%	5%	68%	9%
0.4	52%	62%	98%	4%	68%	8%
0.5	52%	57%	96%	2%	67%	3%
MobileMedia						
	Precision		Recall		F-measure	
K	AHC	FCA	AHC	FCA	AHC	FCA
0.1	85%	85%	100%	100%	92%	92%
0.2	85%	85%	100%	100%	92%	92%
0.3	93%	93%	93%	93%	93%	93%
0.4	93%	93%	93%	93%	93%	93%
0.5	96%	96%	89%	89%	93%	93%

following reasons. Firstly, a function supported by the classes of *code-topic 2* may be shared between the *UseCase* and *Sequence* features, as *Sequence* feature is "an interaction diagram to model the behavior of use cases by describing the way groups of objects interact to complete the task"[1]. Secondly, these two classes are linked to other classes due to false-positive similarity may be computed by VSM, as AHC depends on VSM to compute textual similarity between classes.

Table 5.12 presents an example of a feature implantation obtained by our approach. This implementation represents a set of *ECUs* associated to the *UseCase* feature. We notice that most of the classes presented in this table belong to the same package (argouml.uml.ui.behavior.use_cases) and the term UseCase is apart of their names.

5.8.3 Threats to Validity

We identify two threats of validity to our approach. The first threat is that developers may not use the same vocabularies to name source code identifiers across product variants. This would mean that textual matching at source code level to determine minimal disjoint sets of classes and code-topics identification would be affected. Nonetheless, 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. Consequently, there are common terms between identifiers across product variants. In case of having product variants with different vocabulary for source code identifiers, we can rely

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

Table 5.11 : Examples of code-topics

Code-Topic 1
argouml.uml.diagram.use_case.ui@StylePanelFigUseCase
argouml.uml.ui.behavior.use_cases@ActionNewExtendExtensionPoint
argouml.uml.ui.behavior.use_cases@ActionNewExtensionPoint
argouml.uml.ui.behavior.use_cases@ActionNewUseCase
argouml.uml.ui.behavior.use_cases@ActionNewUseCaseExtensionPoint
Code-Topic 2
argouml.uml.diagram.sequence.ui@FigActivation
argouml.uml.diagram.sequence.ui@FigBirthActivation
argouml.uml.diagram.sequence.ui@SelectionMessage
argouml.uml.diagram.sequence.ui@TempFig
sequence2.diagram@FigMessageSpline
argouml.uml.diagram.use_case.ui@ActorPortFigRect
argouml.uml.diagram.use_case.ui@FigMyCircle

Table 5.12 : Example of a feature implementation

Feature Name: UseCase
argouml.uml.ui.behavior.use_cases@ActionAddExtendExtensionPoint
argouml.uml.ui.behavior.use_cases@ActionNewActor
argouml.uml.ui.behavior.use_cases@ActionNewExtendExtensionPoint
argouml.uml.ui.behavior.use_cases@ActionNewExtensionPoint
argouml.uml.ui.behavior.use_cases@ActionNewUseCase
argouml.uml.ui.behavior.use_cases@ActionNewUseCaseExtensionPoint
argouml.uml.diagram.use_case.ui@StylePanelFigUseCase
argouml.uml.ui.behavior.use_cases@PropPanelActor
argouml.uml.ui.behavior.use_cases@PropPanelExtend
argouml.uml.ui.behavior.use_cases@PropPanelExtensionPoint
argouml.uml.ui.behavior.use_cases@PropPanelInclude
argouml.uml.ui.behavior.use_cases@PropPanelUseCase
argouml.uml.ui.behavior.use_cases@UMLExtendBaseListModel
argouml.uml.ui.behavior.use_cases@UMLExtendExtensionListModel
argouml.uml.ui.behavior.use_cases@UMLExtendExtensionPointListModel
argouml.uml.ui.behavior.use_cases@UMLExtensionPointExtendListModel
argouml.uml.ui.behavior.use_cases@UMLExtensionPointLocationDocument
argouml.uml.ui.behavior.use_cases@UMLExtensionPointUseCaseListModel
argouml.uml.ui.behavior.use_cases@UMLIncludeAdditionListModel
argouml.uml.ui.behavior.use_cases@UMLIncludeBaseListModel
argouml.uml.ui.behavior.use_cases@UMLIncludeListModel
argouml.uml.ui.behavior.use_cases@UMLUseCaseExtendListModel
argouml.uml.ui.behavior.ues_cases@UMLUseCaseExtensionPointListModel
argouml.uml.ui.behavior.use_cases@UMLUseCaseIncludeListModel
.....

on natural language processing tools to find matching between different vocabularies that have the same semantic.

The second threat is that our approach assumes that features are implemented by source code classes. However, features also can be implemented by methods, especially in the case of small-scale systems and in applications that are implemented by procedural languages. Our approach is easily adapted to work with source code methods and procedures, as our strategies to improve IR-based feature location are applicable to support any granularity of source code.

5.9 Conclusion

In this chapter, we have presented an approach to improve the effectiveness of IR namely LSI, as a feature location technique in a collection of product variants. Our approach followed two strategies to achieve this improvement: reducing IR search spaces (feature and source code spaces) and reducing the abstraction gap between feature and source code levels. Firstly, we analyzed commonality and variability distribution across product variants to reduce features and source code of a collection of product variants into a minimal disjoint set of features and their corresponding minimal disjoint set of classes. Secondly, we introduced the concept *code-topic* to reduce the abstraction gap between feature and source code levels. We made a comparison between two techniques to identify code-topics: formal concept analysis (FCA) and agglomerative hierarchal clustering (AHC).

In our experimental evaluation using two case studies of different domains and sizes, we demonstrated that our approach outperforms the conventional application of LSI as well as the most recent and relevant work on the subject in terms of *precision, recall* and *F-measure* [Xue *et al.*, 2012]. We also showed that advantages and disadvantages of FCA and AHC to identify *code-topics*. Moreover, we showed that using AHC for identifying code-topics significantly improves the recall values of LSI-based feature location with a minor decrease in the precision compared to FCA.

CHAPTER 6

Feature-Level Change Impact Analysis Based on Feature Location

Preamble

In this chapter, we propose a feature-level CIA approach based on feature location to study the impact of changes made to implementation of features obtained from product variants. Given a change proposal, the goal of our approach is to allows SPL's manager conducting change management from his point of view by computing a ranked list of features and providing quantitative measures for this purpose. In section 6.1, we motivate our approach. In section 6.2, we give a general overview of feature-level CIA process. We detail the proposed approach steps in sections 6.3, 6.4 and 6.5. In section 6.6, we show experimental results and effectiveness of the proposed approach. Finally, section 6.7 concludes the chapter.

6.1 Introduction

Before committing to the source code changes made to the implementation of features obtained from product variants, SPL's manager may need to understand the change from his point of view rather than the technical details. Often, this point of view is related to market issues and takes into account economic considerations. Feature-level CIA allows for conducting change management from a SPL manager's point of view, as a feature is an agreement between all stakeholders (including managers) about what SPL should do. It provides a sound basis to judge whether the change is worth the effort [Passos *et al.*, 2013], or if inevitable, which features should be changed as a consequence. Consequently, it allows SPL's manager to decide which change strategy should be executed based on affected features. Additionally, feature-level CIA detect the introduction of undesired interactions between feature implementations.

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. As we have seen in the state-of-the-art, the CIA is seldom considered at the feature level for changes made to the source code level. Most the existing approaches performs CIA at the source code level with few approaches completed at other levels. There is only one approach that performs CIA at the feature level however it does not support change management form SPL manager's point of view, as we have seen [Revelle *et al.*, 2011].

In this chapter, we propose a feature-level CIA based on Formal Concept Analysis (FCA). This approach represents an application of feature location. In fact, feature-level CIA mainly relies on the identified traceability links between features and their implementing source code classes in order to determine which features will be impacted for a change induced at the source code level. The proposed approach 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 the ranked list has a degree to be affected representing the feature priority to be checked by maintainers. Additionally, we propose two metrics to support feature level CIA: impact degree and changeability assessment metrics. The former is used to measure to which degree a specific feature can be affected. The latter is used to measure the percentage of features that are affected. This metrics give SPL manager a general overview about the affected features to help him for change management. Our CIA approach provides the following benefits:

1. Performing CIA at the feature level for changes induced at class level before a change is implemented.
2. Quantitative measures to evaluate the ease of implementing a proposed change where there is often more than one change that can solve the same problem.

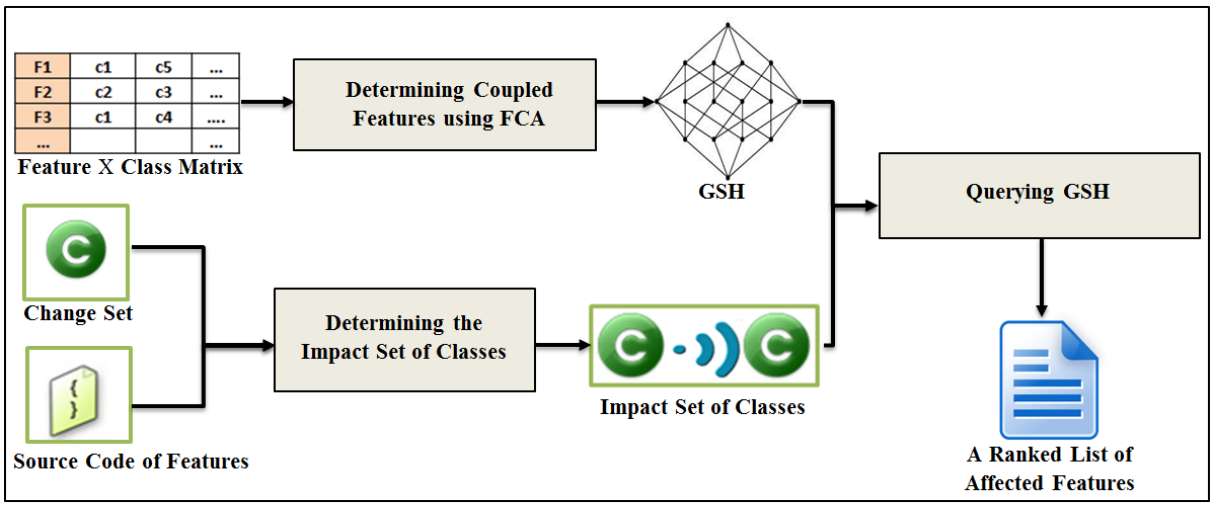

Figure 6.1 : Main Steps of Our Feature-Level CIA Approach.

6.2 Feature-Level CIA Process

When source code classes that contribute to implementing a feature are changed during the maintenance, the change may be propagated to the neighbor classes which the changed classes are coupled to. Classes that are strongly coupled to the modified classes are the most likely to be impacted. For example, suppose that a given class (*C1*) inherits the class (*C2*). If any method declared in C2 is changed, this change may impact the methods defined in `C1`. This is because the inheritance relationship propagates the impact of change from super class to the derived class. As a feature's implementation may span multiple classes and it also may be shared between features, a change may lead to impact the implementation of other features. Therefore to perform feature-level CIA, we should first determine the impact set of classes of a given change set of classes and then determine coupled features. In our approach, 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 [Apel et Beyer, 2011].

Figure 6.1 gives an overview of the proposed approach. This approach relies on three main steps: (i) computing the impact set of classes for source code changes, (ii) analyzing coupled features using FCA, (iii) querying the generated GSH to compute a ranked list of affected features. As shown in the Figure 6.1, the two first steps are executed in parallel. The first step takes as input the change set of classes (CSC) and source code of a given set of feature(s). The objective of this step is to compute the impact set of classes that could be affected by changes made to the source code (i.e., CSC) of features. Of course, this impact set also contains the change set members. The second step takes as input a feature-to-class traceability matrix. In this matrix, each feature is linked to its implementing classes. Column and row labels respectively refer to feature and class names. The matrix values refer to which feature is implemented by which class. The objective of this step is to build the GSH corresponding to a formal context obtained by feature-to-class traceability matrix. This GSH is used to analyze and

determine coupled features. Finally, the third step takes the output of the first two steps (i.e., the impact set of classes and resulting GSH) to compute a ranked list of affected features.

6.3 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 for CIA purpose [Shyam et Chris, 1994] [Churcher et Shepperd, 1995][Sun *et al.*, 2011]:

1. **Inheritance relationship:** when a class inherits attributes and methods from another class.
2. **Method call:** when a method of one class calls a method of another class. For example, when a changed method is called by a method from different a class.
3. **Attribute access:** when a class accesses an attribute of another class. For example, when a class accesses an changed attribute belonging to different class.
4. **Shared attribute access:** when two classes access the same attribute of another class. For example, classes (*A* and *B*) accesses the same attribute (*AT*) that belongs to the class *C*. Class *A* updates the value of *AT* then *B* reads the written value. For some reason, the call statement of AT in class *A* is removed. Then, such a change impacts the class *B* indirectly.

Each one of these interdependencies works as a coupling connection to propagate different kinds of source code changes. Therefore, we depend on them to determine the impact set of classes. To capture these interdependencies, we build a dependency matrix. Columns and rows are identical and represent all source code classes of all features. Matrix values refer to coupling strength between classes. This matrix is constructed by statically analyzing the source code of features through building an abstract syntax tree (AST) [Fluri *et al.*, 2007]. We consider the implementation of all features together as a single application, and then we parse the source code of this application to build AST. This tree is traversed to determine coupled classes based on interdependencies mentioned above. After building such a matrix, classes that are coupled to the change set classes constitute members of the impact set.

This step represents CIA at the source code level. In the literature, there many approaches that can be used to perform this step [Bixin *et al.*, 2012]. We use such a CIA only as a step (not as a research goal) to perform feature-level CIA. Therefore, this step is flexible to be implemented by any existed approach supporting CIA at the source code level.

6.4 Determining Coupled Features Using FCA

As stated before (see section 6.2), determining coupled features is essential for feature-level CIA. In this step, we rely on FCA for this purpose. This is because it allows analyzing and visualizing 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 GSH concepts. Therefore, we build the formal

Table 6.1 : 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

context where objects are features and attributes are classes. The relation between a feature (F) and a class (C) refers to that F is implemented by C. Such a formal context represents the feature-to-class traceability matrix which is an input of this step (see Figure 6.1). According to this definition of the formal context, we can obtain a GSH containing concepts that are composed of a set of features sharing a set of classes. The generated GSH represents a hierarchical organization of features and classes, so that a certain concept inherits its extent (features) from its descendants (sub-concepts) and its intent (classes) from its ascendants (super-concepts). Such a GSH represents dependencies between features and classes (feature coupling).

Table 6.1 is an example of the formal context to be analyzed where objects = $\{F1, F2, F3, F4, F5, F6, F7\}$ and attributes = $\{C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12\}$. Figure 6.2 shows the corresponding GSH of the formal context shown in Table 6.1. Based on this GSH and FCA definitions (see section 2.3 of the preliminaries chapter), we notice the following observations:

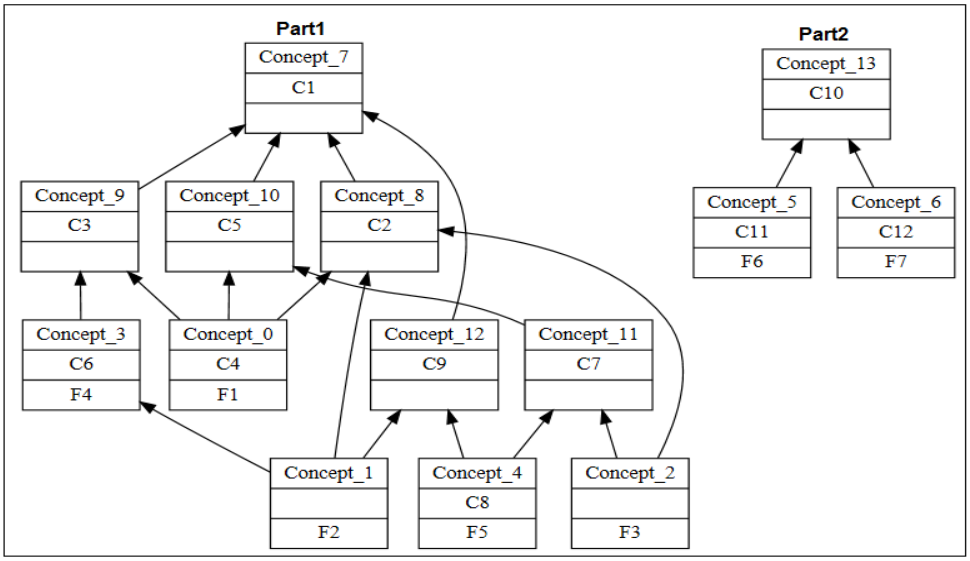

Figure 6.2 : The GSH for the Formal Context of Table 6.1.

- The impact of changes made to classes of GSH concepts that are located at the top of the GSH is propagated to all extents (features) of GSH concepts. 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. This is due to the fact that classes of concepts located at the top are shared among all or most of GSH concepts. Based on this observation, we can find exactly which classes should not be modified or impacted as much as possible during the maintenance because they may lead to risk of changing the implementation of all features, and hence increasing the maintenance costs. Moreover, it is important during regressing testing to pay attention to these classes because they represent core classes of implementation of features obtained from product variants.

- The impact of changes made to classes of GSH concepts that are located at the bottom is local. For example, if *C8* at *Concept_4* belongs to the impact set, only $F5$ will be affected. Based on this observation, we can determine which classes have less impact on system features. 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.

- By descending vertically throughout the GSH, 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 be only composed of $\{F3, F5\}$.

- The features of GSH concepts that are downwardly reachable from jointly changed classes have a high probability to be affected. For example, assuming that the impact set is composed of $\{C2, C3, C5\}$, then $F1$ has a higher probability to be 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 GSH, we can determine groups of features that are isolated from others. These groups may constitute subsystems that are maintained by specific teams. The GSH help to determine required maintenance teams based on the affected subsystems. By referring to the Figure 6.2, we can notice that features are organized into sub-systems (Part1 and Part2) based on the dependency between their implementations. Imagine that the impact set of classes consists of classes (*C10, C11*), and hence the affected features are F6 and F7 of part2. If these features are entrusted to a certain team in the organization, a product manager can exclude other maintenance teams and ask only the interested team to execute the change.

6.5 Querying GSH for Determining a Ranked List of Affected Features

In this section, we present how to determine and rank the affected features for a given change proposal based on the GSH generated in the previous step.

6.5.1 Determining the Affected Features

After generating the GSH of features and classes in the previous step, we use the impact set of classes as a query to retrieve affected features by enquiring this GSH. The retrieved features represent two subsets of affected features, following the two steps below:

- **Step I**: locating a set of GSH concepts that have as intent (excluding inherited intent) one or more of classes that are impacted. The extents of these concepts represent the first subset of affected features.

- **Step II**: determining all downwardly reachable concepts from concepts determined in the step I. The extents of these concepts represent the second subset of affected features.

To perform step I mentioned before, we propose Algorithm 3 to locate the GSH concepts having as intent one or more impacted classes. This algorithm takes as input the impact set of classes (ISC) and the GSH (*CL*). The algorithm returns as output a list of GSH concepts (*CON*) that one or more of their intent elements belong to the ISC set. This list is used as input for the step II.

Algorithm 3: LocatingAffectedConcepts

```
   Input: ISC, CL
1  //ISC: Impact Set of Classes, CL: GSH
   Output: CON // a list of concepts associated with ISC
2  CON ← ϕ
3  foreach CO from 1 to |CL|   // CO is the current concept do
4      if (is intent of the concept CO has one or more of ISC members) then
5          add(CON, CO)
6  return CON
```

To perform step II mentioned before, we propose the Algorithm 4 based on depth-first search (*DFS*) to determine a set of GSH concepts that are downwardly reachable from concepts obtained in the step I. Algorithm 4 takes as input a list of concepts computed in step I (*CON*) and the GSH to be queried (*CL*). Lines 1-11 check each concept in *CON* in turn so that each one becomes the root of a new tree of the DFS. The extents of concepts that constitute such a tree represent all affected features for changes made to intent (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* concepts. LC_F may contain some concepts that do not have own extent and LC_F also can contain redundant concepts due to the overlap between DFS's trees. Therefore, lines 12-15 remove concepts having only inherited extents using the function (*RemovingConHavingEmptyExt()*) and also redundant concept using the function (*RemovingRedundantConcepts()*). Line 16 extracts the extent of LC_F concepts using the function *ExtractingExtent()*. The extents of these concepts represent all the affected features (AF).

6.5.2 Ranking the Affected Features

Based on the observations mentioned in section 6.4, we notice that the GSH hierarchically organizes features according to their degree of impaction for a given change proposal. Based on this hierarchy, we propose two metrics to support our feature-level CIA: *Impact Degree Metric* (*IDM*) and *Changeability Assessment Metric* (*CAM*).

Algorithm 4: LocatingAffectedFeatures

```
Input: CON, CL
Output: AF // list of affected features
LC_F ← φ // a list of concepts downward reachable from CON
foreach j from 1 to |CON| do
    ConceptStack ← CON[j]
    while (ConceptStack is not empty) do
        Co ← pop(ConceptStack)
        AdjCos ← getAdjacentConcepts(Co, CL)
        if (AdjCos is empty) then
            LC_F ← Co
        else
            push(AdjCos)
            LC_F ← Co
foreach i from 1 to |LC_F| do
    if Extent(LC_F[i]) is empty then
        RemovingConHavingEmptyExt( LC_F, i) // remove the concept i from LC_F
RemovingRedundantConcepts (LC_F)
AF ← ExtractingExtent(LC_F)
return AF
```

6.5.2.1 Impact Degree Metric (IDM)

IDM is used to measure the degree to which the implementation of a given feature can be affected. Features having high *IDM* values, the functional requirements provided by these features have a high probability to be affected. Therefore, such a metric may give a SPL's manager an indicator about impacting certain features. They may represent mandatory features or critical features in terms of some aspects related to, for example, economic considerations. We think that IDM is useful for this purpose by giving an indicator about to what extend such features can be impacted and help SPL's manager for making decisions. A list of affected features may contain features whose implementations are in fact not impacted (false-positive features) but features having higher *IDM* values are expected to be actually affected. We propose the following equation for *IDM*:

$$IDM(F) = \frac{|\{I\} \bigcap \{ImpactSetOfClasses\}|}{|\{I\}|} \times 100\% \tag{6.1}$$

In Equation 6.1, F is an affected feature while I is the intent (classes) of a GSH concept having F as an extent and also I includes all intents inherited in a top-down manner from that concept. Thus, we need to compute the inherited intents of GSH concepts (LC_F) having the affected features. This is performed using the DFS algorithm to compute all upwardly reachable concepts from each concept in LC_F. This algorithm is similar to the Algorithm 4, and for prohibiting repetition it is not written.

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.

6.5.2.2 Changeability Assessment Metric (CAM)

CAM is a metric for determining the percentage of features that are affected by a given change. It describes the changeability of 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 to use the following equation for CAM:

$$CAM = \frac{\#Affected\_Features}{\#All\_Features} \times 100\% \tag{6.2}$$

In Equation 6.2, *Affected_Features* represent a set of features that are potentially affected by a given impact set of classes. $All\_Features$ represent all features (including affected feature and others). *CAM* values take a range [0, 1]. If the computed CAM value is high, this means that features (resp. their implementation) is more sensitive for a given change proposal and vice versa.

The value of *CAM* metric is strongly dependent on the coupling between features. Features that are coupled to the implementation of feature(s) being modified are the most likely to be affected, and hence the obtained CAM value is high. However, features that are not coupled to other features can be changed almost in isolation, and hence CAM value for any change proposal will be low. With reference to Figure 6.2 and considering that the impact set is composed of $\{C3, C5, C6\}$, we find the *IDM* and *CAM* values of this impact set as shown in Table 6.2. The columns (*Concept_No, Features* and *Rank*) show respectively a set of concepts having affected features, their affected features and the rank of these features according to IDM values. From Table 6.2, we notice that *F4* has the highest *IDM* value, so its implementation is checked first by maintainers. Also, Table 6.2 shows that *CAM* for this impact set is equal to (71%). This means that most of the features will be affected by the given impact set.

6.6 Experimental Evaluation

In this section, we provide an experimental evaluation of our feature-level CIA approach to demonstrate its feasibility. We have applied it to core assets of three different case studies: ArgoUML-SPL[1],

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

Table 6.2 : Impact results for {C3, C5, C6} changes.

Concept_No	Features	$\lvert\{I\}\rvert$	$\lvert\{I\}\bigcap\{ISC\}\rvert$	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	

Table 6.3 : Subject core assets and their respective information.

Core Assets	#Features	#Classes
ArgoUML-SPL	8	515
MobileMedia	6	28
BerkeleyDB-SPL	25	227

MobileMedia[2] and BerkelyDB-SPL[3]. These case studies were described in the preliminaries chapter (see section 2.4) . Table 6.3 summarizes relevant information of these core assets in terms of number of features and source code classes. In the coming subsections, we present respectively evaluation measures, effectiveness of our approach and threats to validity.

6.6.1 Evaluation Measures

We relied on three measures from information retrieval, namely *precision, recall* and *F-measure* to evaluate our CIA approach [Salton et McGill, 1986][Hattori *et al.*, 2008]. *Precision* measures the accuracy of the estimated affected features according to the actual affected features. *Recall* measures to what degree the estimated affected features covers the actual affected features. 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 see that precision also quantifies affected features that actually are not affected (false-positive). Also, recall quantifies the features that are affected but not identified (false-negatives). In our study, we use the following definitions of *precision, recall* and *F-measure*.

$$Precision = \frac{|\{EAF\} \bigcap \{AAF\}|}{|\{EAF\}|} \times 100\% \tag{6.3}$$

$$Recall = \frac{|\{EAF\} \bigcap \{AAF\}|}{|\{AAF\}|} \times 100\% \tag{6.4}$$

$$F\_measure = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}} \times 100\% \tag{6.5}$$

In the equations above, EAF represents the estimated affected features retrieved by our CIA approach, while AAF represents the actual affected features. To compute AAF, we manually trace all feature implementations to identify the actual affected set of features for a given impact set of classes. Higher *precision, recall* and *F-measure* mean better results. This means that the EAF represents all features actually affected and nothing else. All measures have values within [0,1]. If the EAF has a

[2]http://www.ic.unicamp.br/ tizzei/mobilemedia/
[3]http://www.fosd.de/FeatureVisu/

high precision, this means that maintainers spend less time and effort to locate affected features. If the EAF has a high recall, this gives maintainers the confidence that all of affected features have been considered.

6.6.2 Results and Effectiveness

Table 6.4 summarizes results obtained by our CIA approach. Columns describe respectively: change set of classes (*CSC*), the size of CSC (|*CSC*|), the total number of estimated affected features (|*EAF*), *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 6.4, 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 classes in spite of the fact that it is not affected by this change. Such a case indicates features that are implemented by false-positive classes are actually not affected. Based on precision values, we notice that our approach reduces 60% of maintainers' burden to locate change effects. The precision value can be improved by filter out the false-positive classes. However, this task increases the maintainers' burden. Even though we use another approach to compute the impact set of classes, the result of that approach should be checked by maintainers to determine false-positive elements as with any CIA approach.

Recall values shown in Table 6.4 are high where these values take a range between 75% and 100%, and in most cases they reach 100%. The reason that hinders our approach achieving 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 approach gives a good compromise between precision and recall, as these values are high taking a range between 67% and 100%.

Table 6.4 shows the changeability of features (CAM) 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 features. For *ArgoUML-SPL*, all changes made to its features affect almost half of its features. This is due to *ArgoUML-SPL*'s features being loosely coupled. They seem 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 changed classes and the GSH corresponding to this case study, we find that one of the changed classes (*EnvironmentImpl*) is located at the top of the GSH. 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, we notice that these values quantify the changeability of the extracted features against each change proposal. This allows SPL's manager to select the change strategy with lowest possible CAM value.

Table 6.4 : Precision, Recall and F-measure of our CIA Approach.

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%

6.6.3 Threats to Validity

We identify three issues that constitute limitations of our study and impact the results. The first two issues are related to CIA at the source code level, which is outside of scope of our concern in this work.

- Firstly, our approach studies many different source code changes, such as deletion of a class and changes made to a class body. Deleting the dead source code and outputting statements are also studied, but actually, such changes do not impact system features. As a result, this will cause false-positive impact set of classes.
- Secondly, our approach does not consider classes that are not neighbors of modified classes although these classes may be impacted, and hence degrades the recall values.
- Finally, the change set is randomly selected, which may not be the actual proposed changes in subject systems. Therefore, this may affect the results.

6.7 Conclusion

In this chapter, we have proposed a feature-level CIA approach to study the impact of changes made to source code elements of features obtained from product variants. This approach leveraged the traceability links between given features and their implementing source code elements to perform CIA at the feature level.

Our approach helps SPL managers to conduct change management from their point of view. Therefore, it returns a ranked list of affected features, as a feature is better understood by a SPL's

manager. Also, two metrics are suggested to support change management from a manager's point of view: *IDM* and *CAM*. The form is used to compute to which degree a given feature is affected and compute the changeability of all features.

The proposed approach takes, as input, a change set of classes, feature-to-class traceability matrix and source code of features. It computes a ranked list of potentially affected features. In the proposed approach, we employed structural and feature couplings. Structural coupling was used to determine the impact set of classes of a given change request. Feature coupling was used determine features coupled to the implementation of feature(s) being modified. We used formal concept analysis to determine coupled features. 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 widely used metrics on the subject (*precision, recall* and *F-measure*).

CHAPTER 7

Toward Reverse Engineering of SPLA from Product Variants and Feature to Architecture Traceability

Preamble

In this chapter, we propose an approach to support reverse engineering SPLA taking advantage of existing software product variants. This reverse engineering task represents the second application of feature location in our work. In this reverse engineering task, we focus on identifying mandatory components and variation points of components as an important step toward SPLA. In section 7.1, we motivate the need for our approach. In section 7.2, we present an overview of the process of identifying mandatory components and variation points of components. We detail the process phases in sections 7.3 and 7.4. Section 7.5 shows the experimental evaluation to demonstrate the feasibility of the proposed approach. Finally, this chapter ends with a conclusion in section 7.6.

7.1 Introduction

Software Product Line Architecture (SPLA) constitutes the backbone of successful SPL. It provides a coherent picture of the different components that must be developed and to equip them with generic interfaces that can be used throughout the different products [Linden *et al.*, 2007]. It is used to develop several concrete architectures for SPL's products. To that end, it must encompass commonality and variability of requirements of SPL's products.

Developing SPLA from scratch is a costly task because it should encompass the components realizing all the mandatory and varying features in a particular domain [Pohl *et al.*, 2010]. SPLA can be reverse engineered from software product variants by exploiting commonality and variability across product variants and extracting components from their source code. Reverse engineering SPLA from software product variants has not been considered in the literature. As mentioned in the respective state-of-the-art, most existing works support reverse engineering software architecture from single existing software system.

In this chapter, we propose an approach to contribute for building SPLA taking advantage of existing product variants. Our contribution is twofold:

- Identifying mandatory features and variation points (VPs) of features of a given collection of product variants as this identification represents the main source of commonality and variability of SPLA.
- Exploiting the commonality and variability in terms of features to identify mandatory components and VPs of components, as this organization of components represent an important step toward building SPLA.

We propose a set of algorithms to identify mandatory features and VPs of features. Then, we adapt a component extraction approach (called, *ROMANTIC*) proposed by my team to extract components from the implementation of feature groups (mandatory features and VPs of features) instead of extracting components from the entire source code of each software product independently, as this approach is usually used [Chardigny *et al.*, 2008][Kebir *et al.*, 2012b]. This adaptation is based on traceability links between features and source code. These links allow determination of the implementation of each feature group where *ROMANTIC* can be applied. Also, these links are used to link features to the extracted components. Such links between features and components represent variability traceability links to bind variability at architectural level (SPLA).

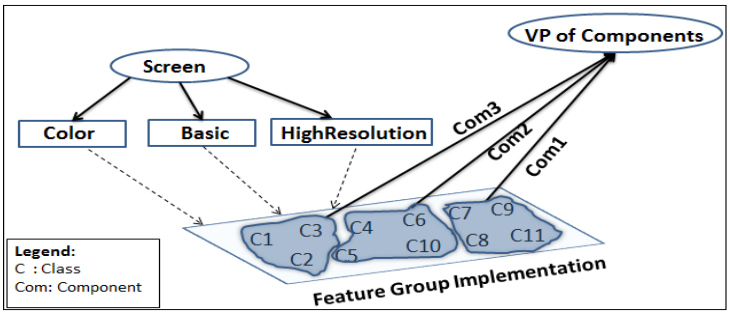

Figure 7.1 : An Example of Extraction VP of Components.

7.2 Overview of the Proposed Approach

Reverse engineering SPLA from existing software product variants involves the extraction components from source code of these variants as the component represents the main building unit in SPLA. The extracted components, as mentioned in the state-of-the-art chapter (see section 4.2), should be organized as mandatory components and a set of VPs of components. This organization is driven by commonality and variability across product variants in terms of features. Therefore, we should first identify mandatory features (commonality) and VPs of features (variability) across product variants. Then, we need to exploit this commonality and variability to identify mandatory components and VPs of components. In our approach, components are extracted from the implementation of each group of features (i.e., mandatory features group and VPs of features). Figure 7.1 shows an example of components extraction from the source code classes of a VP of features (*Basic, Color* and *HighResolutio*). Such a method for component extraction allows to extract components as groups because the VP in SPLA is a group of components [Pohl *et al.*, 2010]. Also, it allows to create only one VP for each VP at feature level for more flexibility and evolvability of SPLA.

Figure 7.2 shows an overview of the process of commonality and variability identification at feature and architectural level. This process takes three inputs: (i) product configurations (PCs) in which each is a set of features provided by some product in product variants; (ii) the implementation of all features in product variants; and (iii) feature descriptions. Our process steps are organized into two phases. In the first phase, we identify mandatory features and VPs of features. In the second phase, we identify mandatory components and VPs of components.

The first phase takes as input PCs and feature descriptions. It first starts by identifying mandatory features of product variants. Next, configurations are pruned by excluding mandatory features. In this way, the remaining features in each configuration represent the optional features. Then, the optional features are organized into four group types : *AND-Group, XOR-Group, OR-Group* and *OP-Group* (the semantic of each group is defined in section 2.1.3 of the preliminaries chapter). We exclude from the members of identified groups before identifying the members of the next type of groups. Each identified group represents a VP at the feature level.

In the second phase, we identify mandatory components and VPs of components. This phase starts by extracting components from source code classes of each VP of features and from mandatory features group. Next, each feature is linked to its corresponding component(s) for binding common-

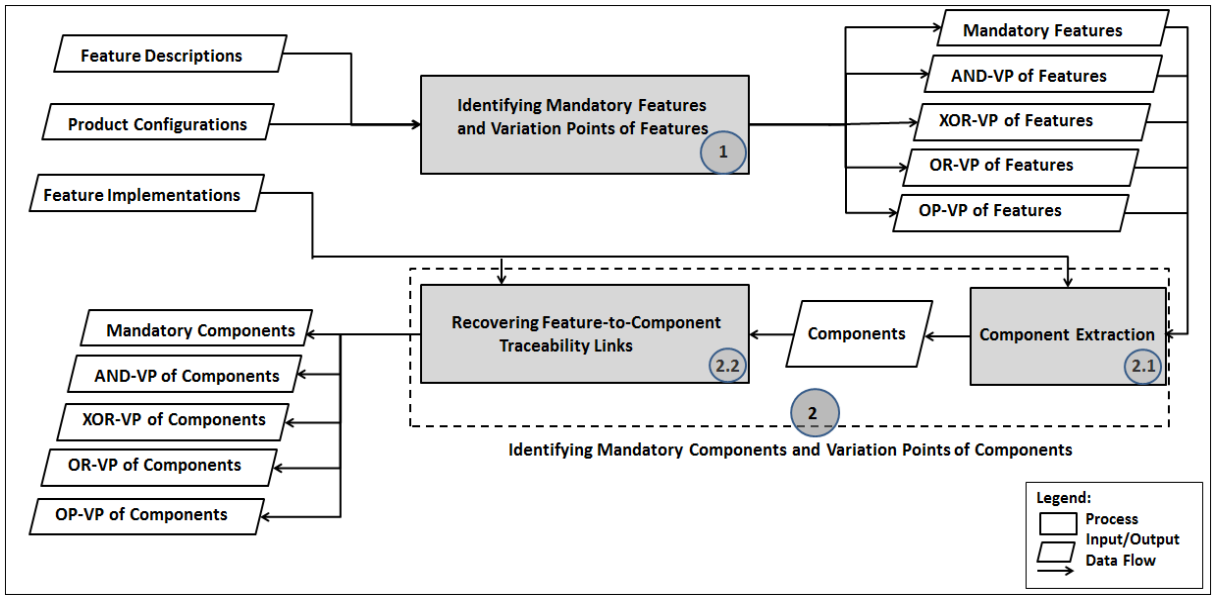

Figure 7.2 : An overview of the Process of Identifying VPs of Components and Features.

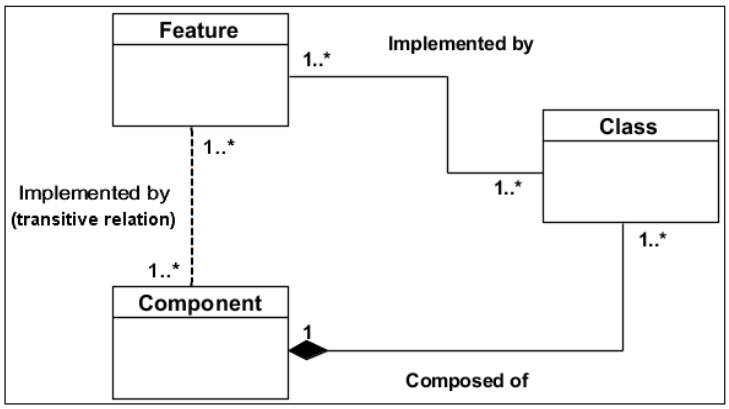

Figure 7.3 : Representation of Linking Feature and Component by MetaModel.

ality and variability at the architectural level in order to identify mandatory components and VP of components. Such linking is performed by exploiting transitive relation between feature and component, as both can be linked to source code classes. Figure 7.3 shows a meta model depicting the transitive relation between features and components through source code classes. Below, we describe these phases in more details.

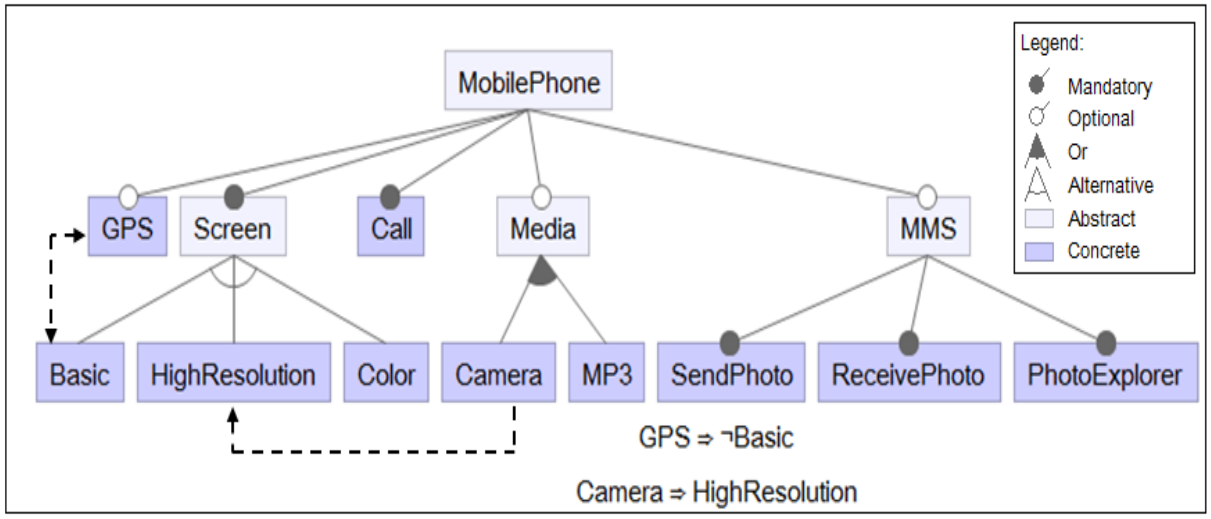

Figure 7.4 : FM of MobilePhone SPL Adapted from [Benavides *et al.*, 2010].

7.3 Identifying Mandatory Features and Variation Points of Features

As commonality and variability in terms of features represent the main source of commonality and variability in SPLA, we need to identify this commonality and variability of given product configurations of product variants. To this end, we propose a set of algorithms. The algorithms are designed by considering the definition of each type of VPs. The algorithms also are applied in a specific sequence so that firstly, mandatory features are identified then AND-Groups, XOR-Groups, OR-Groups and OP-Group. The idea behind this sequence is that features that always exhibits regular behavior in product configurations (like mandatory features and AND-Groups) should be firstly identified in order to reduce the search space for identifying other types of VPs that exhibit irregular behavior.

7.3.1 Basic Definitions

To explain our proposed algorithms, we use FM shown in Figure 7.4 in order to generate valid product configurations (PCs). We treat the generated configurations as product variant configurations. Table 7.1 shows all possible valid configurations that can be generated from FM in Figure 7.4. We use as column labels the shortest distinguishable prefix of the feature names (e.g. *Co* for *Color* feature shown in Figure 7.4). The check symbol (✓) refers to the features provided by each configuration. Before presenting our algorithms, we start by defining the key concepts which are shared among the proposed algorithms.

Definition 5 (Feature List) *Feature list (FL) is a list of all unique features in all product configurations. All unique features provided by FM shown in Figure 7.4 represent an example of FL.*

Definition 6 (Feature Set) *Feature set is a tuple* $FS = [sf, \overline{sf}]$ *where* sf *and* $\overline{sf}$ *are respectively the set of selected and non-selected features of a given product configuration* (PC_i) *[Haslinger* et al., *2011].*

Table 7.1 : product configurations of mobile phone SPL

Product	Ca	S	R	P	B	H	Co	Ca	M	G
PC_1	✓				✓					
PC_2	✓					✓				
PC_3	✓						✓			
PC_4	✓					✓				✓
PC_5	✓						✓			✓
PC_6	✓					✓		✓		
PC_7	✓				✓				✓	
PC_8	✓					✓			✓	
PC_9	✓						✓		✓	
PC_{10}	✓					✓		✓	✓	
PC_{11}	✓					✓		✓		✓
PC_{12}	✓					✓			✓	✓
PC_{13}	✓						✓		✓	✓
PC_{14}	✓					✓		✓	✓	✓
PC_{15}	✓	✓	✓	✓	✓					
PC_{16}	✓	✓	✓	✓		✓				
PC_{17}	✓	✓	✓	✓			✓			
PC_{18}	✓	✓	✓	✓		✓				✓
PC_{19}	✓	✓	✓	✓			✓			✓
PC_{20}	✓	✓	✓	✓		✓		✓		
PC_{21}	✓	✓	✓	✓	✓				✓	
PC_{22}	✓	✓	✓	✓		✓			✓	
PC_{23}	✓	✓	✓	✓			✓		✓	
PC_{24}	✓	✓	✓	✓		✓		✓	✓	
PC_{25}	✓	✓	✓	✓		✓		✓		✓
PC_{26}	✓	✓	✓	✓		✓			✓	✓
PC_{27}	✓	✓	✓	✓			✓		✓	✓
PC_{28}	✓	✓	✓	✓		✓		✓	✓	✓

Thus $PC_i.sef \cap PC_i.\overline{sef} = \Phi$ *and* $PC_i.sef \cup PC_i.\overline{sef} = FL$.

An example of a FS is the product configuration PC_{18} = [{ *Call, SendPhoto, ReceivePhoto, PhotoExplorer, HighResolution, GPS*}, {*Basic, Color, Camer, MP3*}] in Table 7.1. In this configuration, $PC_{18}.sef$ is composed of {*Call, SendPhoto, ReceivePhoto, PhotoExplorer, HighResolution, GPS*} while $PC_{18}.\overline{sef}$ is composed of {*Basic, Color, Camer, MP3*}.

Definition 7 (Feature Set Table) *Feature set table (FST) is a collection of FSs. Each row in this table represents a PC so that for every product* P_i *we have* $PC_i.sef \cup PC_i.\overline{sef} = FL$. *Table 7.1 is an example of FST.*

The functions *add()* and *remove()* during the algorithms presented below mean respectively adding and removing the second argument to/from the values of the first argument. Notice that we assume a pass-by-reference argument semantic throughout the algorithms.

7.3.2 Identifying Mandatory Features

The behavior of mandatory features imposes existing these features in all products variants. Based on this behavior, we propose algorithm 5 for identifying mandatory features. The algorithm takes as input *FST* and FL of a given collection of product variants, and returns as output a set of mandatory features (*MAF*). The algorithm works by conducting a textual matching between feature names of

given product configurations (steps 1-3) to find features that are part of all configurations. These features represent the mandatory features. The function (*smallestProducts()*) returns feature names of the smallest product in terms of provided features. After identifying mandatory features, we prune FST and FL (step 4) by excluding mandatory features from their contents using the function (*Prune()*). The time complexity of the algorithm 5 is *O(n)* where *n* is the number of available configurations.

Algorithm 5: Identifying Mandatory Features

Input: FST, FL
Output: MAF //**Ma**ndatory **F**eatures
1 Set MAF ← smallestProduct(FST)
2 **foreach** *i from 1 to* |*FST*| **do**
3 MAF ← MAF ∩P_i.Sef
4 Purne(FST, FL, AGF)
5 return MAF

7.3.3 Identifying AND-Variation Points of Features

The behavior of an AND-Group of features across product configurations seems like an atomic set whose members always appear or disappear together. Considering features f_1 and f_2 as an AND-Group, this means that for each product configuration (PC_i), if $f_1 \in PC_i.sef$ *iff* $f_2 \in PC_i.sef$ and if $f_1 \in PC_i.\overline{sef}$ *iff* $f_2 \in PC_i.\overline{sef}$.

Based on the regular behavior of AND-Groups, we propose Algorithm 6 to determine their members. The main data structure used during this algorithm is a multiset. The algorithm starts by intersecting pair-wisely all selected feature sides ($PC_i.sef$) for all product configurations in *FST* (steps 3-5). This intersection aims at finding candidate sets (*Intersect_sef*) that their members (features) may appear together. Of course, not each set in *Intersect_sef* represents AND-Group. Therefore, we pair-wisely check the elements of each set in *Intersect_sef* against the semantic of the AND-Group (steps 6-11). Any pair of features that respects this semantic is put in *AGF*, otherwise it is rejected. As a result, each set in *AGF* is either an AND-Group only consisting of two members or a pair of features that complies to a *require* constraint like *HighResolution* and *Camera* features in Figure 7.4. Step 12 uses feature descriptions (*ApplyFeaDes()*) to filter out pairs of features in *AGF* that comply to a *require* constraint. The idea behind using feature descriptions is that features that belong to the same group should have common terms in their descriptions. Therefore, a pair of features that does not share terms in their description is rejected. As a result, the remaining pairs in *AGF* represent only AND-Groups only consisting of two features. Steps (13-16) merge together pairs in *AGF* that have transitive relations to form AND-Groups of three or more members.

Actually, the members of an obtained AND-Group after merging may belong to two or more AND-Groups. For example, according to our algorithm, features (F1 to F6) in Figure 7.6.A will be identified a single AND-Group, as they appear and disappear together. However, these features belong to two AND-Groups. To deal with such a situation, we again rely on feature description (step 17). The idea behind using feature descriptions is the same as we explained previously. Next, we prune *FST* and *FL* by removing AND-Group members (step 18). By referring to Table 7.1 and applying this algorithm, we

Algorithm 6: Identifying AND-Group Variation Points

Input: FST, FL, Feature Description
Output: AGF // **A**ND-**G**roups of **F**etures

```
1  Intersect_sef ← Φ // Multisets of features
2  AGF ← Φ // Multisets of features
3  foreach i from 1 to |FST| − 1 do
4      foreach j from i+1 to |FST| do
5          intersect_Sef ← P_i.sef ∩ P_j.sef
6  foreach Set S ∈ Intersect_sef, with |S| > 1 do
7      foreach i from 1 to |S| − 1 do
8          foreach j from i+1 to |S| do
9              Set temp, add(temp, S_i), add(temp, S_j)
10             if temp isn't partially present in any FST's products then
11                 add(AGF, temp)
12 AGF ← ApplyFeaDes(AGF)
13 foreach i from 1 to |AGF|-1 do
14     foreach j from i+1 to |AGF| do
15         if (AGF_i and AGF_j are transitive) then
16             merge(AGF_i, AGF_j)
17 AGF ← ApplyFeaDes(AGF)
18 Purne(FST, FL, AGF)
19 return AGF
```

found that there is only one AND-Group consisting of {*SendPhoto, RecievePhoto and PhotoExplorer*} and FM in Figure 7.4 confirms that also. The time complexity of the algorithm 6 is $O(n^5)$ where n is the number of available configurations.

7.3.4 Identifying XOR-Variation Points of Features

According to the semantic of XOR groups, the behavior of members of a XOR-Group across product configurations entails the existence of only one member in $PC_i.sef$ while the remaining members in $PC_i.\overline{sef}$. Based on this behavior, we propose Algorithm 7 to identify members of XOR-Groups. The main data structures used through the algorithm are *Multiset* and *HashMap* (step 1). In steps (2-6), we assume that each feature (F) in FL is a member of a XOR-Group. This means that $PC_i.\overline{sef}$ where F belongs to $PC_i.sef$ includes all other members of that group. Therefore, if F belongs to many product configurations in FST, we obtain many sets corresponding to F. Of course, not all elements of these sets have exclusive relations with F. Therefore, we intersect all these sets to filter out irrelevant elements (features) as much as possible. After the intersection, F corresponds to a unique set. Then, F and the unique resulting set is kept as an entry in a HashMap called *ExRe*, where F and its corresponding set respectively represent the *key* and *value* of this HashMap. Each entry in *ExRe* represents

Algorithm 7: Identifying XOR-Group Variation Points

```
   Input: FST, FL, Feature Descriptions
   Output: XGF (XOR-Groups of Fetures)
1  XGF ← φ, //multiset    ExRe ← φ //HashMap
2  foreach F ∈ FL do
3      foreach i from 1 to |FST| do
4          if F ∈ P_i.sef then
5              intersect ← intersect ∩ P_i.sef
6      ExRe.put(F, intersect) //ExRe.put(key, value)
7  foreach entry En ∈ ExRe do
8      Set RHS ← En.getValue(),  count ← 0
9      Set CurExRe ← RHS, add(CurExRe, En.getKey())
10     if !( CurExRe ∈ XGF ) then
11         foreach feature f ∈ RHS do
12             Set temp1 ← CurExRe, remove (temp1, f)
13             if temp1 ⊆ ExRe.getValue(f) then
14                 count++
15     if count = |RHS| then
16         add(XGF, CurExRe)
17 XGF ← ApplyFeaDesForEx(XGF)
18 XGF ← ApplyFeaDesForGroup(XGF)
19 Purne(FST, FL, XGF)
20 return XGF
```

excluded-relation that takes the following formate *[F ⇔ set of features]*. *F* represents the left-hand side (LHS) of a *excluded-relation* while the corresponding set of *F* represents the right-hand side (RHS). Figure 7.5 shows *excluded-relations* obtained from our illustrative example.

1-[Basic ↔ Camera, Color, GPS, HighResolution]
2-[Camera ↔ Basic, Color]
3-[GPS ↔ Basic]
4-[HighResolution ↔ Basic, Color]
5-[Color ↔ Basic, Camera, HighResolution]

Figure 7.5 : All Excluded-Relations of FM in Figure 7.4.

The elements of RHS of an entry in ExRe are a combination of features so that this combination may only consist of a XOR-Group's members or it may include members of other groups. Therefore, steps (7-14) check each entry in *ExRe* against the definition of XOR-Group. Considering that an entry (*En*) in `ExRe` is a XOR-Group, this means that each feature (*f*) in the RHS of *En* must appear as a LHS of another *excluded-relation* (*Ex1*) and RHS of *Ex1* must contain all `En's` features except *f*. By

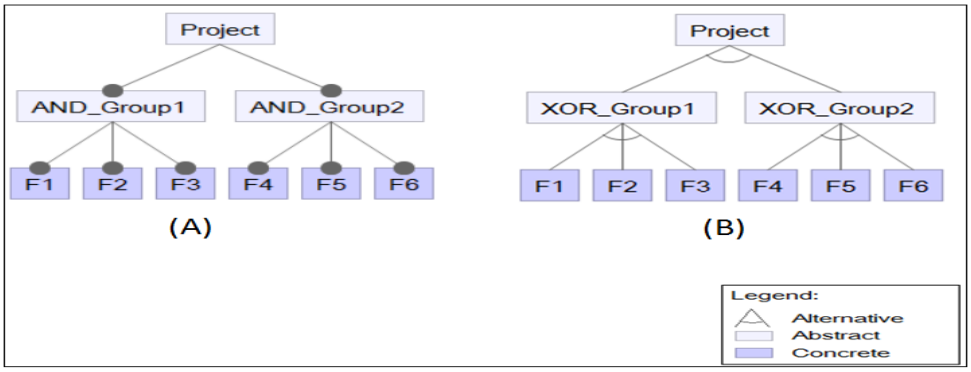

Figure 7.6 : An Example of Nested AND-Group and XOR-Group.

applying these steps, entries 2, 3 and 4 in Figure 7.5 are identified as candidate XOR-Groups and put in XGF while others are rejected (steps 15-16). XGF may contain groups only consisting of two members (e.g., entry 3 in Figure 7.5). Such groups are not necessary to be XOR-Groups. They may be pairs of features that comply to an *exclude-constraint* such as *GPS* and *Basic* features in Figure 7.4. Therefore in step 17, we use feature descriptions (*ApplyFeaDesForEx()*) to identify and then remove such groups from XGF. Also, XGF may contain a group that has the semantic of a XOR-Group but in fact their members can not be aggregated as a single XOR-Group. Such a group appears due to one of the following reasons.

- Firstly, the members of such a group belong to two or more XOR-Groups. For example, according to our algorithm, features (F1 to F6) in Figure 7.6.B are identified as single XOR-Group, as inclusion any feature of them excludes others (considering that their parents marked as mandatory). However, these features belong to two XOR-Groups. To deal with such a situation, we again rely on feature description (step 18) to re-organize members of each identified XOR-Group.

- Secondly, such a group may appear due to cross tree constraints (i.e., require and exclude constraints). For example, entry 2 in Figure 7.5 is identified as XOR-Group in spite of *Camera* feature belongs to OR-Group. This happened due to the cross tree constraint between *Camera* and *HighResolution* (see Figure 7.4) which generate an alternative relation between *Camera*, *Color* and *HighResolution* features. To detect such a group, we use feature descriptions like the previous situation. If we fail to re-organize members of such a group into one or more XOR-Groups provided that the organizing process consumes all members, we reject such a group. Consequently, the remaining exuded-relations in Figure 7.5 represent XOR-Groups.

Finally, in step 19 we prune *FST* and *FL* by removing members of XOR-Groups. The time complexity of the algorithm 7 is $O(n^4)$ where n is the number of available configurations.

7.3.5 Identifying OR and OP Variation Points of Features

Although the remaining features in FST are only OR-Groups and OP-Groups, the identification of their members is a challenge because the behavior of members of these groups is arbitrarily. For the identification of OP-Groups, we completely rely on feature descriptions. We consider features that have common keywords in their descriptions as belonging to the same OR-Group. In this way, we can identify OR-Groups. The time complexity of the proposed algorithm for identifying members of OR-VP is $O(F^2)$ where F is the number of remaining features in FL.

After identifying OR-Groups, the remaining features represent the members of a single OP-Group.

7.4 Identifying Mandatory Components and Variation Points of Components

In this phase, we extract components from source code of feature groups and then we identify mandatory components and VPs of components based on commonality and variability identified in the previous phase.

7.4.1 Component Extraction

As mentioned earlier, a component is a well-known building unit to build an architectural view of a system [Allier *et al.*, 2011][Oussalah, 2014]. In our approach, a component is extracted based on ROMANTIC approach as a cluster of classes [Chardigny *et al.*, 2008]. Each cluster is extracted based on the quality metrics defined in ROMANTIC (see Appendix C). ROMANTIC is applied to source code classes implementing feature groups: mandatory features, AND-Group, XOR-Group, OR-Group and OP-Group. Such an application of ROMANTIC respects the components organization in SPLA (i.e., mandatory components and VPs of components). We obtain source code classes that implement each feature group using our IR-based feature location approaches. Components that are extracted from the implementation of mandatory features constitute the mandatory components, while components extracted from each identified feature group constitute members of a VP in SPLA. In our approach, each VP at feature level corresponds to only one VP at the architectural level. Additionally, such an application of ROMANTIC reduces source code space where ROMANTIC can be applied so that this space only represents feature implementations. This means that we filter out source code elements related to the platform used to execute the product variants and hence the extracted components only implement features.

The implementation of feature groups may have shared source code classes across different feature groups. Such classes are not specific to a certain feature group and they may implement features that have cross cutting behavior across all other features. Therefore, we determine such classes and then we apply ROMANTIC to these classes as a group. The extracted components represent a component group called *Shared-Com*. They are not specific to a certain VP in SPLA and the selection of these components for products development depends on the selection of their associated features.

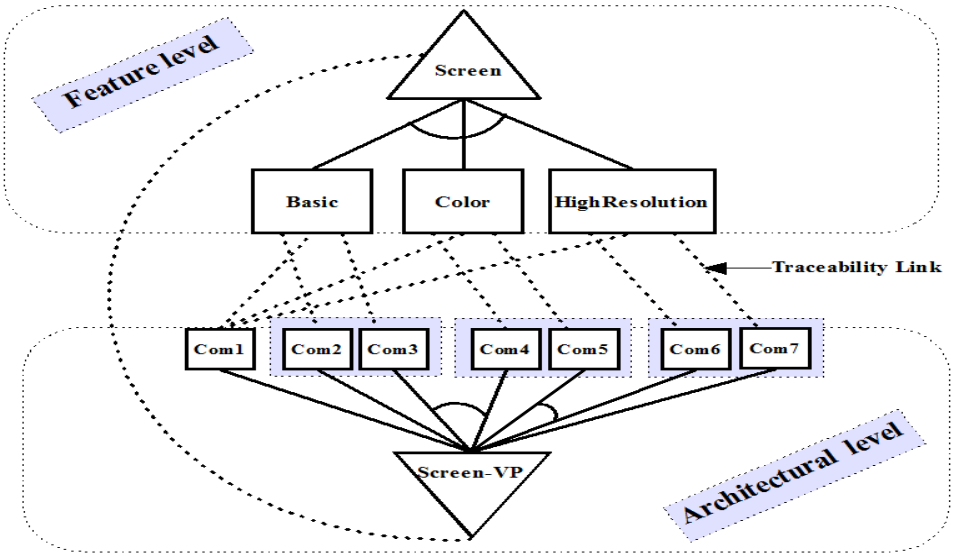

Figure 7.7 : An Example for Linking VP of Features to VP of Components.

7.4.2 Recovering Feature-to-Component Traceability Links

From the previous step, we notice that components of a VP do not necessary have the semantic of that VP. For example, consider that *Color* and *HighResolution* are two features which belong to XOR-Group, and components extracted from the source code implementing this group are [com1, com2, com3 and com4]. Also, assume that the first three components implement `Color`, while `com4` implements `HighResolution`. In this case the relation among the first three components is not exclusive, although they belong to an exclusive VP. This is because the mapping between components and features is many-to-many [Pohl *et al.*, 2010]. This means that a feature's implementation may be scattered over more than one component inside a VP and also a component may implement more than one feature (refer to Figure 7.3). This mapping should be considered during the creation of VPs through establishing explicit links between features and their corresponding components. These links are useful for:

- Binding variability in SPLA. This means that such links determine a combination of components inside each VP so that each combination represents a variant of that VP. For example, [com1, com2 and com3] mentioned above represents only one variant. Determining variants of each VP in SPLA is important to specify the constraints among components. Choosing a variant of components depends on customer choices at feature level. Therefore, such links bind variability in SPLA.

- Determining the components that implement each feature. This facilitates and automates the derivation of architectures of SPL products from the SPLA and makes the architecture derivation process driven by customers' needs (features).

Such traceability links between features and components can be established by exploiting the transitive relation between features and components through source code classes. Based on meta model shown in Figure 7.3, a feature is implemented by set of classes and a component is composed of set of classes. Therefore, classes are a shared element between features and components, which allow them (components and features) to link together.

After linking features and components, it is normal to have shared components between all features belonging to the same VP due to the many-to-many mapping between features and components. Therefore, these components are not specific to a certain feature but they are related to the parent of those features and form a parent of VP of components in SPLA. To clarify this idea, see Figure 7.7. In this figure, there is a VP at feature level of type XOR called *Screen* consisting of three features (*Basic, Color, HighResolution*). The corresponding VP at architecture level of *Screen* is *VP1* which consists of seven components [*Com1 to Com7*]. Highlighted boxes represent pairs of components so that each pair excludes others. Dotted lines refer to traceability links between each feature and their implementing components. From this figure, we notice that *Com1* is shared among three features so it is not specific to a certain feature. Therefore, *Com1* forms the parent of remaining components in *VP1*.

7.5 Experimental Evaluation

In this section, we present an experimental evaluation to demonstrate the feasibility of our approach. We apply it to three case studies: *MobilePhone, ArgoUML-SPL* and *MobileMedia*. MobilePhone is only a FM without codebase so it is only used to evaluate the proposed algorithms. This FM is an enhanced version of FM of our illustrative example (see Figure 7.4). We design this FM by considering all VPs types which are not present in real case studies. This FM contains four VPs from different sizes and types. In addition, it includes two cross tree constraints, two optional features and two mandatory features. Figure 7.8 shows MobilePhone's FM. ArgoUML-SPL and MobileMedia are presented and detailed in the preliminaries chapter.

Table 7.2 presents mandatory features and VPs of features provided by MobilePhone's FM, ArgoUML-SPL's FM and MobileMedia's FM respectively. We label each VP based on its type, for purpose of comparison. We organize our evaluation into two parts. In the first part, we validate the algorithms used to identify mandatory features and VPs of features. In the second part, we validate the identification of mandatory components and VPs of components.

7.5.1 Validating the Identification of VPs of Features

7.5.1.1 Evaluation Measures

As a base for evaluation, we match each VP identified by our approach with its corresponding VP in the focused FM. This matching is measured by using three metrics inspired from information retrieval field, namely *precision*, *recall* and F-measure [Baeza-Yates et Ribeiro-Neto, 1999] [Hattori *et al.*, 2008]. *Precision* measures the accuracy of identified members of a VP according to the relevant members of that VP. *Recall* measures to what degree the identified members of a VP cover the relevant members of that VP. F-measure is used to find the best possible compromise between *precision* and *recall* values. The relevant members of each VP are determined by FM. All measures have values within $[0,1]$. For a

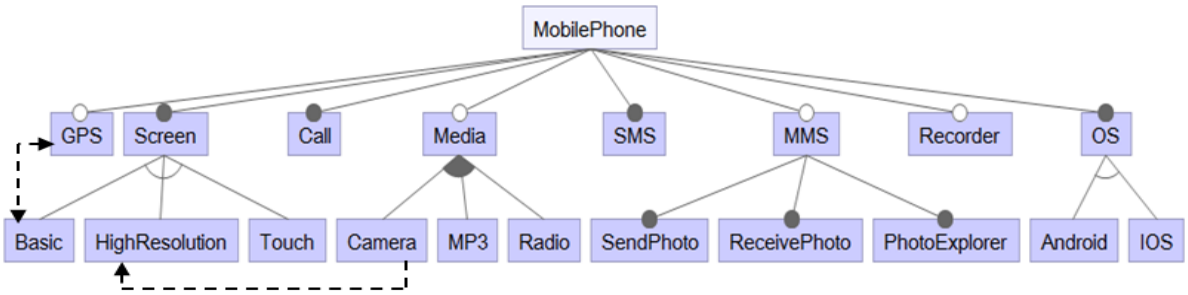

Figure 7.8 : FM of MobilePhone.

Table 7.2 : Mandatory features and VPs of features for MobilePhone, ArgoUML-SPL and MobileMedia

MobilePhone	
Mandatory Features and VPs	**Members**
Mandatory	*Call, SMS*
AND-Group VP1	*SendPhoto, ReceivePhoto, PhotoExplorer*
XOR-Group VP1	*Basic, HighResolution, Touch*
XOR-Group VP2	*Android, IOS*
OR-Group VP1	*Camera, MP3, Radio*
OP-Group VP1	*GPS, Recorder*
ArgoUML-SPL	
Mandatory	*Class*
OR-Group VP1	*State, Activity, UseCase, Collboration, Deployment, Sequence*
OP-Group VP1	*Logging, CognitiveSupport*
MobileMedia	
Mandatory	*Album Management, Basic Photo Operations*
OR-Group VP1	*Copy Photo, Edit Photo Label, SMS Transfer*
OP-Group VP1	*Exception Handling*

given VP_i, if precision equals 1, this means that all identified members are relevant. However, some relevant members might not be identified. If recall equals 1, this means that all relevant members of VP_i are identified. However, some identified members might not be relevant. If F-measure equals 1, this means that all relevant members are identified and only them. Our proposed algorithms aim to achieve high precision, recall and F-measure. We propose the equations 7.1 and 7.2 to adapt *Precision, Recall* and F-measure in our context. In these equation, *IM_VP* and *RM_VP* refers to **I**dentified **M**embers and **R**elevant **M**embers of a given **VP**.

$$Precision(VP_i) = \frac{|IM\_VP_i \cap RM\_VP_i|}{|IM\_VP_i|} \times 100\% \tag{7.1}$$

$$Recall(VP_i) = \frac{|IM\_VP_i \bigcap RM\_VP_i|}{|RM\_VP_i|} \times 100\% \tag{7.2}$$

$$F-measure(VP_i) = \frac{2}{\frac{1}{Recall_i} + \frac{1}{Precision_i}} \times 100\% \tag{7.3}$$

7.5.1.2 Results

We randomly generate three sets of configurations from MobilePhone's FM, two sets from ArgoUML-SPL's FM and three sets from MobileMedia using FeatureIDE tool[1] (well-known SPL tool). Each generated set has a different size and it also covers all features shown in its corresponding FM. The size of MobilePhone's sets is as follows: sets from 1 to 3 respectively (224, 112 and 10) configuration. The size of ArgoUML-SPL's sets is as follows: sets from 1 to 2 respectively (256 and 8) configuration. The size of MobileMedia's sets is as follows: sets from 1 to 3 respectively (16, 8 and 5) configuration. The *set1* in all case studies represents all possible configurations can be generated from its corresponding FM. All generated product configurations are available in the Appendix B.

Table 7.3 shows obtained results by applying our algorithms to product configurations generated from FMs of case studies considered. In this table, we present *precision*, *recall* and *F-measure* for each VP identified and average precision, recall and F-measure for all identified VPs corresponding to each set of configurations. Highlighted rows refer to VPs that have been identified by our algorithms but they actually do not exist in FMs of cases studies considered (false-positive VPs).

In *MobilePhone*, we notice that the proposed algorithms give 100% *Precision*, 100% *Recall* and 100% *F-measure* for each VP in both Set_1 (containing all possible configurations) and Set_3 (containing only 10 configurations). This means that the number of available configurations is not only the dominant parameter in the proposed algorithms to achieve 100% for *Precision*, *Recall* and *F-measure*. This also means that the proposed algorithms can identify all relevant members (and only those members) for each VP in the case of having a few configurations, provided that these configurations have a high diversity of feature combinations to detect the behavior of each member of each VP. This fact is confirmed by the results of Set_2. InSet_2, although the number of configurations is large (half of all possible configurations that can be generated from the FM), the proposed algorithms identify false-positive VP (OR-Group VP2) and fail to identify VP present in MobilePhone's FM (XOR-Group VP1). This is because the configurations of Set_2 lack the diversity to distinguish the members of each VP Similarly in MobileMedia case study, the proposed algorithms give 100% *Precision*, 100% *Recall* and 100% *F-measure* for each VP in both Set_1 (containing all possible configurations) and Set_3 (containing only 5 configurations). This is because these configurations have a high diversity of feature combinations to distinguish the members of each VP. In Set_2, although the number of configurations is half of all possible configurations, the proposed algorithms fail to identify two VPs (*OR-Group VP1*, *OP-Group VP1*) and return two false positive VPs (*XOR-Group VP1*, *XOR-Group VP2*). This is because the configurations of Set_2 lack the diversity to distinguish the members of each VP.

In ArgoUML-SPL, the identified VPs from Set_1 and Set_2 have the same *precision*, *recall* and *F-measure* values, in spite of the fact that *set1* represents all possible configurations, while Set_2 rep-

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

Table 7.3 : Precision, recall and F-measure of identified mandatory features and VPs of features for case studies considered.

MobilePhone						
Set_1**: No. Configurations = 224 (All possible configurations)**						
	Precision	Recall	F-measure	Average Precision	Average Recall	Average F-measure
Mandatory	100%	100%	100%	100%	100%	100%
AND-Group VP1	100%	100%	100%			
XOR-Group VP1	100%	100%	100%			
XOR-Group VP2	100%	100%	100%			
OR-Group VP1	100%	100%	100%			
OP-Group VP1	100%	100%	100%			
Set_2**: No. Configurations = 112**						
Mandatory	100%	100%	100%	71%	71%	71%
AND-Group VP1	100%	100%	100%			
XOR-Group VP1	0.0%	0.0%	0.0%			
XOR-Group VP2	100%	100%	100%			
OR-Group VP1	100%	100%	100%			
OR-Group VP2	0.0%	0.0%	0.0%			
OP-Group VP1	100%	100%	100%			
Set_3**: No. Configurations = 10**						
Mandatory	100%	100%	100%	100%	100%	100%
AND-Group VP1	100%	100%	100%			
XOR-Group VP1	100%	100%	100%			
XOR-Group VP2	100%	100%	100%			
OR-Group VP1	100%	100%	100%			
OP-Group VP1	100%	100%	100%			
ArgoUML-SPL						
Set_1**: No. Configurations = 256 (All possible configurations)**						
Mandatory	100%	100%	100%	58%	67%	62%
OR-Group VP1	67%	100%	80%			
OP-Group VP1	0.0%	0.0%	0.0%			
Set_2**: No. Configurations = 7**						
Mandatory	100%	100%	100%	58%	67%	62%
OR-Group VP1	67%	100%	80%			
OP-Group VP1	0.0%	0.0%	0.0%			
MobileMedia						
Set_1**: No. Configurations = 16 (All possible configurations)**						
Mandatory	100%	100%	100%	100%	100%	100%
OR-Group VP1	100%	100%	100%			
OP-Group VP1	100%	100%	100%			
Set_2**: No. Configurations = 8**						
Mandatory	100%	100%	100%	25%	25%	25%
OR-Group VP1	0.0%	0.0%	0.0%			
OP-Group VP1	0.0%	0.0%	0.0%			
XOR-Group VP1	0.0%	0.0%	0.0%			
XOR-Group VP2	0.0%	0.0%	0.0%			
Set_3**: No. Configurations = 5**						
Mandatory	100%	100%	100%	100%	100%	100%
OR-Group VP1	100%	100%	100%			
OP-Group VP1	100%	100%	100%			

resents a very small number of configurations. This is due to the fact that the FM of ArgoUML-SPL just offers two types of VPs (*OR-Group VP1*, *OP-Group VP1*) and the configurations of these sets have a high diversity to distinguish members of each VP. This means that feature descriptions play important role to identify these VPs. We also notice that the algorithms have failed to identify *CognitiveSupport* and *Logging* (their label is *OP-Group VP1* in Table 7.2) features as a VP of type *OP-Group* but they are identified as *OR-Group*. This is because these features share keywords with all other features as they have a crosscutting behavior in all ArgoUML-SPL features. Therefore, (*CognitiveSupport* and *Logging*) are identified as members of *OR-Group VP1*. This leads to degradation of *Precision* and *Recall* values of *OP-Group VP1* and *OR-Group VP1* as shown in Table 7.3.

7.5.2 Validating the Identification of VPs of Components

In the ideal case, we should compare the identified mandatory components and VPs of components with already existing mandatory components and VPs of components of SPLA, which is recovered from software product variants. Unfortunately, reverse engineering SPLA from product variants has not been considered in the literature. Therefore, we demonstrate the performance of our approach through the interpretation of the obtained results.

We generate 7 products corresponding to the Set_2 of ArgoUMl-SPL (see Table 7.3). These products provide all features offered by ArgoUML-SPL's FM. Also, we use 5 existing releases of MobileMedie. The configurations of these releases correspond to the Set_3 of MobileMedia (see Table 7.3). All considered variants of both case studies are described in preliminaries chapter (see section 2.4).

Table 7.4 shows the identified mandatory components and VPs of components by applying our approach to the 7 products generated from ArgoUML-SPL code base. *Parent Components* column presents components that form the parent of a VP while *Member Components* column presents components that form members of that VP. In this table, we show for each VP at the architectural level its corresponding VP at the feature level. The label of the identified VPs of components are as follows: *OR-VP1* and *OP-VP1*.

For components of *OR-VP1*, we notice that all names of parent components of OR-VP1 share the term "Diagram". This means that they are not specific to a certain feature (UML Diagram) but they may support all UML diagrams. By referring to ArgoUML-SPL's FM, we find out the name of the parent of group of features corresponding to OR-VP1 is "Diagrams". Consequently, our approach can distinguish between parent components and member components. For components of OP-VP1 which corresponds to *CognitiveSupport* and *Logging* features, our approach identifies no parent components for this VP. This is because its components are only extracted from the implementation of only one feature (*CognitiveSupport*) while the *Logging* feature is implemented by external library (Log4J) [Couto *et al.*, 2011]. Thus, the OP-VP1 only consists of components of *cognitiveSupport* feature, and hence they represent only one variant of OP-VP1. Based on the names of these components, we notice that they are specific to *CognitiveSupport* because their names contain "Cr " term which refers to *Critics* supported by *CognitiveSupport* feature, as this feature is used to detect design errors made by end users. For mandatory components which are extracted from the implementation of mandatory feature (Class diagram), the names of these components show that these components implement this feature as their names contain the term "Classdiagram". Of course, there are no parent components for mandatory components because they are extracted from the implementation of

Table 7.4 : Mandatory components and variation points at architectural level for ArgoUML-SPL.

At Architecture Level: OR-VP1	
At Feature Level: OR-Group VP1 (State, Activity, UseCase, Collaboration, Deployment, Sequence)	
Parent Components	**Member Components**
Fig Prop Diagram Action Mode List State	Action Fig Button New Prop State Selection
Fig Diagram State Sequence Model Activity Renderer	UML Model List Fig Prop Case Use
UML Fig Diagram Init Model Action List	Action Fig Mode Iterator Message State Attributes
Go Fig To State Diagram Collaboration Machine	
At Architecture Level: OP-VP1	
At Feature Level: OP-Group VP1 (CognitiveSupport, Logging)	
Parent Components	**Member Components**
No Parent Components	Cr Wiz Child Many Conflict No Gen
	Cr To Name Do By Missing Class
	List To Go Cr Init Wiz
	Cr Resolved Abstract Transitions Name Todo Math
	Wiz Default Step To Renderer Tree Prop
	Cr Goals Dialog Add Param Type To
	Node Decision Knowledge Goal Priority Type
	Cr Go Wiz No Name Oper Invalid
	List Action To Object Tab Do
	Table Cr Checklist To Abstract UML Model
At Architecture Level: Mandatory Components	
At Feature level: Mandatory Features: Class	
Parent Components	**Member Components**
No Parent Components	Fig Style List Class Panel Class diagram Model
	Package Character Port Rect Fig
	Action Show Hide Visibility Stereotype
	Classdiagram Fig Subsystem Association Edge
	Classdiagram Fig Event Class Diagram Edge Object
	Selection Fig Class Node List Math Stereotype
At Architecture Level: Shared-Com	
Cr Without Instance Classifier	
Cr Without Class Component	
Cr Without Instance Node Comp	
Cr Without Instance Classifier Node Collection Iterator	
Cr Without Node Component	
Cr Without Instance Classifier Component	
Cr Interface Without Component	

only one feature (Class diagram). For components of *Shared-Com*, all components of this group related to only *CognitiveSupport* feature as their names include “Cr ”terms. This is expected because this feature has cross cutting behavior through all other features. This means the implementation of this feature is shared among the implementation of other feature groups, which represents the semantic of *Shared-Com* group. Consequently, our approach can extract and determine components that are shared between VPs of components (*Shared-Com*).

Table 7.5 shows the identified mandatory components and VPs of components by applying our approach to the 5 releases of MobileMedia. The structure of this table is the same as the Table 7.4. This table shows two identified VPs of components (OR-VP1 and OP-VP1) and mandatory components. For OR-VP1, the components of this VP is extracted from the implementation of three features (CopyPhoto, EditPhotoLabel and SMS Transfer). This VP has only member components and does not

Table 7.5 : Mandatory components and variation points at architectural level for MobileMedia.

At Architecture Level: OR-VP1	
At Feature Level: OR-Group VP1 (Copy Photo, Edit Photo Label, SMS Transfer)	
Parent Components	**Member Components**
Photo String Screen Controller Thread Base Network	Sms Sender Controller
	Photo View Controller
	Sms Controller Receiver
	Sms Messaging
	Sms Receiver Thread
	Class New System Print Screen Stream Label
	Album Controller
At Architecture Level: OP-VP1	
At Feature Level: OP-Group VP1 (Exception Handling)	
Parent Components	**Member Components**
No Parent Components	Exception Invalid Image Album Format Unavailable Persistence
At Architecture Level: Mandatory Components	
At Feature level: Mandatory Features: Album Management, Basic Photo Operations	
Parent Components	**Member Components**
Photo Screen Album New Add List System	No Members Components
At Architecture Level: Shared-Com	
No Components	

have parent components. The names of these member components refer to features corresponding to that VP (OR-VP1). For example, the terms *sms, messaging, sender, receiver* refer to *SMS Transfer* feature, the term *label* refers to *EditPhotoLabel* feature and the term *Photo* refers to both *CopyPhoto* and *EditPhotoLabel* features. For OP-VP1, this VP correspond to only one feature (Exception handling) so there are no parent components for this VP. Also, this VP consists of only one member component as shown in the table. The name of this component refers to the *Exception handling* feature, as his name includes terms (exception, invalid, unnavigable) that are related to this feature. Table 7.5 shows there is only one mandatory component that is extracted from the implementation of two mandatory features (*Album Management* and *Basic Photo Operations*). This component represents a parent component because it is shared between these features, and hence there are no member components. The name of this component refers to these features together, as its name includes the terms (*Album, Photo, New and Add*) that are related to these features.

7.5.3 Threats to Validity

Our approach identifies commonality and variability at feature level and exploits them to identify mandatory components and variation points of components for SPLA. Thus, the quality of the proposed approach depends on the quality of identifying commonality and variability in terms of features. In the case of having a few configurations with less diversity, we need the expert intervention to validate all VPs of features. The expert's burden depends on the available configurations for product variants. If there are many configurations with high diversity, the expert's burden will be reduced and vice versa.

Sometimes, we can encounter optional features that have crosscutting behavior throughout all SPL's features (e.g., *CognitiveSupport* in ArgoUML-SPL). In such a situation, we also need an expert intervention to identify such features.

7.6 Conclusion

In this chapter, we have proposed an approach to contribute towards supporting reverse engineering SPLA from object-oriented product variants. Our approach focused on identifying mandatory components and variation points of components, as this identification represents an important step toward building SPLA.

As SPLA encompasses commonality and variability at feature level, we first analyze commonality and variability across product variants in terms of features in order to identify mandatory features and variations points of features. This commonality and variability are reflected at architectural level as mandatory components and variation points of components. We have proposed a set of algorithms to identify different types of variation points at the feature level: *AND-Group, OR-Group, XOR-Group* and *OP-Group*. These algorithms were designed based on the semantic of each type of variation point. Then, we relied on a reverse engineering tool called *ROMANTIC* to extract components from source code of product variants. We applied *ROMANTIC* to the implementation of each variation point of features individually. Based on this application of *ROMANTIC*, we identified, for each VP at the feature level, its corresponding VP at architectural level. Next, we exploited the transitive relation between features and components to link them. These links played two roles: implementation link and variability traceability link to bind variability at the architectural level.

In our experimental evaluation using three case studies, we showed that if we have many product configurations with high diversity and precise feature descriptions, the proposed algorithms achieve high precision and recall of identified variation points of features, and hence this leads to identify relevant variation point of components. Also, we proved the effectiveness of our approach for identifying mandatory components and variation points of components.

CHAPTER 8

Conclusion and Future Work

The general goal of this thesis is to support re-engineering product variants into SPLs. Towards that goal, we studied the problem of finding traceability links between features and their implementing source code elements in context of product variants. Then, we addressed how to use such traceability links for the following tasks concerning that goal.

1. Supporting change impact analysis at feature level.

2. Contributing to build Software Product Line Architecture (SPLA).

Traceability links between features and their implementing source code elements are essential for understanding source code of product variants, and then reusing features (resp. their implementations) for developing new products taking advantage of SPLE. Such traceability is also necessary for facilitating and automating new product derivation from SPL's core assets when the re-engineering process is completed. Information Retrieval (IR) techniques are used widely for identifying such traceability links. These techniques deal with product variants as separated and independent entities. However, commonality and variability across product variant should be exploited to improve the process of finding such traceability links.

The implementing source code elements of features obtained from product variants may need to be changed for adapting SPLE context by adding or removing requirements (resp. their source code elements) to meet new needs of customers. In this case, feature-level Change Impact Analysis (CIA) is needed to determine feature(s) that may be impacted for a given change proposal before the change is implemented. It is helpful to conduct change management from a SPL manager's point of view for economic consideration related to the re-engineering process. Feature-level CIA is seldom

considered. Most survey approaches performs CIA at the source code level with a few approaches completed at requirement and design levels.

The development of SPLA from scratch is a costly task because it represents an infrastructure to derive components not only for one architecture but also for all architectures of SPL's products. As a result, existing product variants should be reused as much as possible to support reverse engineering SPLA by extracting components and organizing these components as mandatory and members of variation points. This organization represents an important step toward this reverse-engineering task. In the literature, reverse engineering SPLA from product variants has not been considered and surveyed approaches follow forward engineering way for SPLA development.

8.1 Summary of Contributions

To summarize, the contributions of this thesis are as follows:

- **IR-feature location in a collection of product variants**
  We proposed an approach to locate feature implementations in a collection of product variants. The proposed approach is based on IR. In the approach, we improved the effectiveness of IR-feature location by: (i) reducing IR search spaces (feature and source code spaces) where IR are applied. This is performed by analyzing commonality and variability across product variants using formal concept analysis. (ii) reducing the abstraction gap between feature and source code levels. This is performed by introducing the concept of code-topic as an intermediate level. In our experimental evaluation, we showed that our approach outperforms the conventional application of IR as well as the most recent and relevant work on the subject, in terms of the widely used metrics for evaluation: *precision, recall* and *F-measure*.

- **Feature-level change impact analysis: based on feature location**
  We proposed an approach for studying the impact of changes made to the source code of features obtained from product variants based on formal concept analysis. This approach helps to conduct change management from a SPL's manager point of view, as the feature is an agreement among all stakeholders (including managers) about what the system should do. This allows a SPL's manager to decide which change strategy should be executed, as there is often more than one change that can solve the same problem. Two metrics are suggested for such change management: IDM and CAM. The former is used to measure the degree to which a given feature can be affected. The latter is used to compute the changeability of all features obtained from product variants. Our approach mainly leverages the identified traceability links between feature and source code classes to detect and study change impact at the feature level for changes made at the source code level. Such traceability links propagate the impact of source code changes at the feature level to be studied on that level. In our experimental evaluation, we proved the effectiveness of our approach in terms of the most commonly used metrics on the subject (*precision, recall* and *F-measure*).

- **Toward reverse engineering SPLA from product variants: based on feature location**
  We proposed an approach towards for reverse engineering Software Product Line Architecture (SPLA) from product variants. In this reverse engineering task, we focused on identify-

ing mandatory components and variation points of components as an important step toward this architecture. This organization of components represents the commonality and variability in SPLA, which is driven by commonality and variability at the feature level. Our proposed approach represents the second application of feature location that help us to determine the implementation of feature groups from where components are extracted. Also based on feature location, we established traceability links between features and extracted components to bind variability at the architecture level (SPLA). We proposed a set of algorithms to determine this commonality and variability in terms of features of a given set of product configurations. According to the experimental evaluation, the efficiency of these algorithms mainly depends on the available product configurations and feature descriptions. If we have a large number of configurations with high diversity and precise feature descriptions, our approach achieves high precision and recall of identified mandatory features (commonality) and variation points of features (variability), and hence this leads to identify relevant mandatory components and variation point of components.

8.2 Direct Perspectives

In this section, we discuss some possible extensions of our current work and issues that are not handled.

- **Experiments with large number of case studies:**
  The performance of our proposed approaches depends on the availability of well-documented product variants in terms of feature descriptions and truth ground links between features and their implementing source code elements, which are difficult to collect. Thus, we plan to extend our evaluation by conducting more experiments to better test our approaches.

- **Supporting different granularities of source code:**
  In our work, we assume that a feature is implemented at source code level by classes. However, a feature can be implemented by different low source code granularities (such as, methods, statement), especially in case of small-scale systems. Thus, we plan to extend our approaches to work on granularity below class level, e.g. methods.

- **Extracting feature model:**
  We plan to extend our proposed algorithms to identify commonality and variability in terms of features across product variants in order to extract feature models. The current algorithms determine only mandatory features and feature groups but we need to determine the hierarchical organization of these features to extract the feature model.

- **Feature interactions:**
  "A feature interaction is a situation in which the composition of multiple feature implementations leads to emergent behavior that does not occur when one of them is absent" [Apel *et al.*, 2011]. The emergent behavior can be undesirable and cause failures in SPLs. Feature interactions occur during the product derivation process to combine different features together for generating SPLs. We plan to detect undesired feature interactions during product derivation process.

Publications

I have started my PhD in January 2012 . In the following, we give our publications during this thesis.

- **International conferences**

  1. Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony. *Feature-to-code traceability in legacy software variants*. Euromicro Conference series on Software Engineering and Advanced Applications (SEAA'13), IEEE Computer Society, 2013.
  2. Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony. *Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval*. IEEE International Conference on Information Reuse and Integration (IRI'13), IEEE Computer Society, 2013.
  3. Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony. *Feature-Level Change Impact Analysis Using Formal Concept Analysis*. Software Engineering and Knowledge Engineering (SEKE'26), 2014.
  4. Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony. *Feature Location in a Collection of Product Variants: Combining Information Retrieval and Hierarchical Clustering*. Software Engineering and Knowledge Engineering (SEKE'26), 2014.

- **International workshops**

  1. Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony, Ra'fat Al-msie'deen. *Recovering traceability links between feature models and source code of product variants*. Variability for You(VARY'12), Austria, ACM, 2012.
  2. Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony, Ra'fat Al-msie'deen. *Identifying traceability links between product variants and their features*. International workshop on Reverse Variability Engineering (REVE'1), Italy, 2013.

- **Paper under review**

  1. Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony. *Toward Recovering Component-based Software Product Line Architecture from Object-Oriented Product Variants*. submitted to international software product line engineering conference (SPLC'18).

- **Two demo papers**

  1. Hamzeh Eyal-Salman. Toward Reengineering Legacy Software Variants into Software Product Line: Feature-to-Code Traceability, Doctoral Symposium, ECOOP conference 2013, France, Montpellier.
  2. Hamzeh Eyal-Salman, Abdelhak Seriai, Christophe Dony and Ra'Fat Al-Msie'Deen "Identifying Traceability between Feature Model and Source Code in a Collection of Product Variants ". Workday Group RIMEL LPG Nîmes, December 17, 2012

Part IV

Appendices

APPENDIX A

Implementation Architecture of Proposed Approaches

We have implemented our proposals in JAVA. The implementation architecture consists of two layers: *feature location* layer and *feature location applications* layer (see Figure A.1). The top layer receives three inputs concerning product variants: *product configurations*, *source code* and *feature descriptions*. This layer aims to locate feature implementations in a collection of product variants. Under the feature location layer is the feature location applications layer. This layer is a group of two subsystems called respectively: *feature-level CIA* and *SPLA*. Feature-level CIA receives change request and feature implementations, and returns a ranked list of affected features. SPLA receives feature descriptions, product configurations and feature implementations, and returns mandatory components and variation points of components. Below, we detail each layer as a separated architecture. Each architecture is a set of components with *required* and *provided* interfaces.

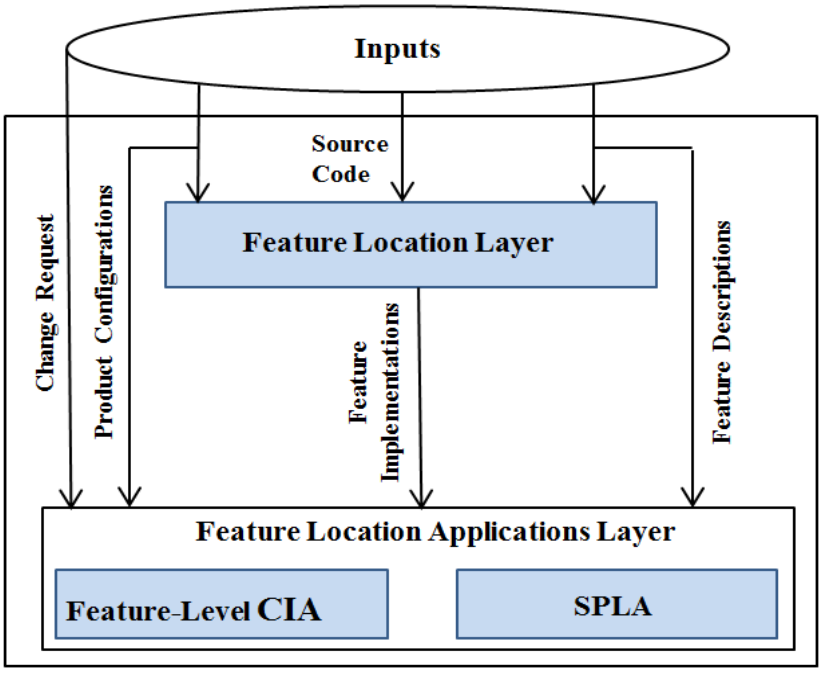

Figure A.1 : Layered Implementation Architecture.

A.1 Implementation Architecture of our Feature Location Approach

Figure A.2 shows the implementation architecture of feature location approach as a set of components. Below, we detail these components according to the order of their use in the feature location process:

- **FCA**. This component implements Formal Concept Analysis (FCA) technique. It provides GSH corresponding to the given formal context. In our work, this component deals with different formal contexts: product configurations, cosine similarity matrix, structural dependency matrix and combination of cosine similarity and structural dependency matrices. The usage of GSH corresponds to these contexts as follows:
  - Analysis variability distribution in terms of features across product variants. For this service, FCA component requires product configurations and provides the corresponding GSH *Minimal Disjoint Sets of Features and Classes Finder* component.
  - Clustering similar classes together. For this service, FCA component requires the mentioned above matrices to provide the corresponding GSH to *Code-Topics Generator* component.

  Most of the existing tools implementing FCA are referenced from the web page of Uta Priss[1].

[1] http://www.upriss.org.uk/fca/fca.html

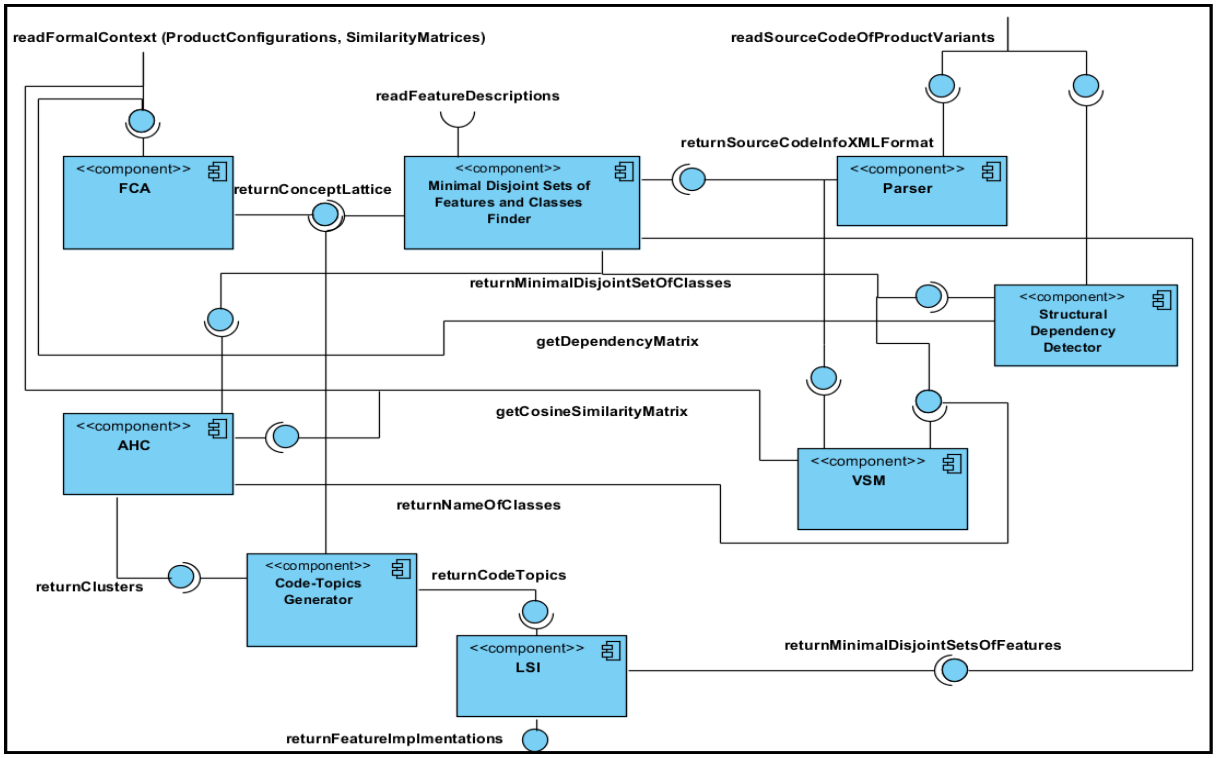

Figure A.2 : Feature Location Implementation Architecture.

For this work, we used the eclipse eRCA [2].

- **Minimal Disjoint Sets of Features and Classes Finder**. This component is responsible for reducing IR search spaces (feature and source code) into minimal disjoint sets of features (resp. source code classes). It requires feature descriptions, source code of product variants in XML format and GSH generated by FCA. It provides minimal disjoint sets of features and classes.

- **ASTParser**. This component is responsible for obtaining all source code information: names of packages, classes, methods, attributes, local variables, method invocation. Also, it is used to obtain structural dependency information, such as inheritance relation, attributes access, method invocation. It saves all this information in XML formate. It requires source code of all product variants and provides source code information in XML format for each variant separately. We used Java parser *JDT/AST* to parse source code of a given collection of product variants. Also, we used *JDOM* parser to save source code information in XML format.

- **VSM**. This component is responsible for computing textual similarity of a given set of classes. It requires a minimal disjoint set of classes. The textual information related to these classes is obtained by calling *ASTParser* component. Then, VSM creates a document for each class containing its textual information. The collection of class documents represents the corpus

[2] https://code.google.com/p/erca/

and query documents. We used the *TD/IDF* formula to assign a weight for each term extracted from class documents. VSM component provides a similarity matrix of these given classes. I have implemented VSM according to the description mentioned in the preliminaries chapter [3].

- **Structural Dependency Detector**. This component is responsible for determining structural dependencies of a given set of classes. It requires a minimal disjoint set of classes. Also, this component builds a data structure to save source code of a given a collection of product variants. This data structure is explored to determine structural dependencies among a given minimal disjoint set of classes. This component provides a structural dependency matrix of these given classes.

- **AHC**. The role of this component is to group similar classes of a given minimal disjoint set of classes into clusters according to agglomerative hierarchical clustering principles. It invokes the VSM components to compute textual similarity, initially among class documents, then between class documents and cluster documents during the clustering process and among only cluster documents. This component provides a set of clusters. Each cluster can be a candidate *code-topic*.

- **Code-Topics Generator**. This component is responsible for identifying code-topics from each minimal disjoint set of classes. It requires FCA and Agglomerative Hierarchical Clustering (AHC) components to cluster similar classes together. It provides a set of code-topics that can be derived from a given minimal disjoint set of classes.

- **LSI**. The role of this component is to compute textual similarity between feature descriptions and code-topics information. It requires both minimal disjoint sets of features and code-topics derived from the minimal disjoint set of classes. To create LSI's corpus, a document is created for each code-topic containing the source code information for that code-topic. Also to create query documents, a document is created for each feature containing the feature description. The *TD/IDF* formula is used to assign a weight for each term extracted from corpus and query documents. We used an external library to implement SVD technique called *JAMA* [4]. Cosine similarity equation is used to measure textual similarity between corpus and query documents. After linking features to their corresponding code-topics, this component also decomposes each code-topic into its classes. This component provides the implementation of each feature as a set of classes. I have implemented LSI according to the description mentioned in the preliminaries chapter [5].

Table A.1 presents statistical source code information of the the feature location approach in terms of number of packages (NOP), number of classes (NOC) and number of line of codes (LOC).

[3] https://code.google.com/p/tool-vsm
[4] http://math.nist.gov/javanumerics/jama/
[5] https://code.google.com/p/tool-lsi

Table A.1 : Statistical source code information of feature location approach.

Approach	NOP	NOC	LOC
Feature Location with IR	21	61	5,221

A.2 Implementation Architecture of our Feature-Level CIA Approach

Figure A.3 shows the implementation architecture of feature-level CIA approach as a set of components. Below, we detail these components according to the order of their use in the feature-level CIA process.

- **Affected Classes Finder**. This component is responsible for determining impact set of classes for a given change set of classes. In this component, we build data structure to save source code of obtained features from product variants. This data structure contains packages, classes of these features. We explore this data structure to find classes that are coupled to the change set according to predefined coupling relations (see section 6.3). We parse the source code of features using JDT/AST. This is performed by building a Java project containing source code of all features. This component provides the impact set of classes.
- **FCA**. The role of this component is to generate a GSH corresponding to formal context that represents the feature-to-code traceability matrix. This component is the same FCA component presented in the architecture of feature location process. We again presented it because it is shared between these architectures.
- **Affected Features Finder**. The role of this component is to determine a ranked list of affected features. It requires the GSH generated by the *FCA* component and impact set of classes obtained by *Affected Classes Generator* component. In this component, we query the GSH by the impact set of classes according to algorithms proposed in section 6.5. It provides a ranked list of affected features as a final output.

Table A.2 presents statistical source code information of the feature level CIA approach in terms of number of packages (NOP), number of classes (NOC) and number of line of codes (LOC).

Table A.2 : Statistical source code information of feature-level CIA approach.

Approach	NOP	NOC	LOC
Feature-Level CIA	7	25	2,223

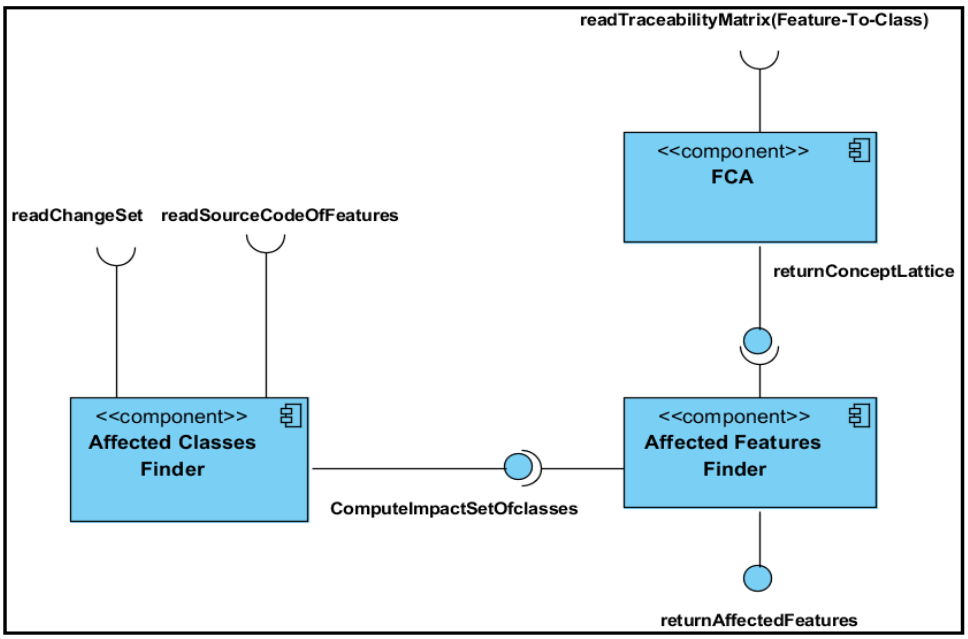

Figure A.3 : Feature-Level CIA Implementation Architecture.

A.3 Implementation Architecture of our SPLA reverse-engineering approach

Figure A.4 shows the implementation architecture of SPLA approach as a set of components. We detail these components according to the order of their uses.

- **Feature-Level Commonality & Variability Analyzer**. This component is responsible for determining mandatory features (commonality) and variation points (VPs) of features (variability) across product variants. It requires product configurations and feature descriptions. In this component, we implement the algorithms proposed in section 7.3. It provides mandatory features, AND-Groups, XOR-Groups, OR-Groups and OP-Groups of features.
- **Feature Group Implementation Finder**. The role of this component is to determine the implementation of feature groups (i.e., VPs and mandatory features). It requires all feature implementations. This implementation consists of all source code classes corresponding to each feature.

  and provides feature group implementations.
- **ROMANTIC**. This component is responsible for extracting components from the implementation of feature groups. It requires a feature group implementation and provides a set of components. We dedicated the Appendix C to detail how ROMANTIC works.
- **Architectural-Level Commonality & Variability Analyzer**. The role of this component is to determine mandatory components and VPs of components. It requires components extracted

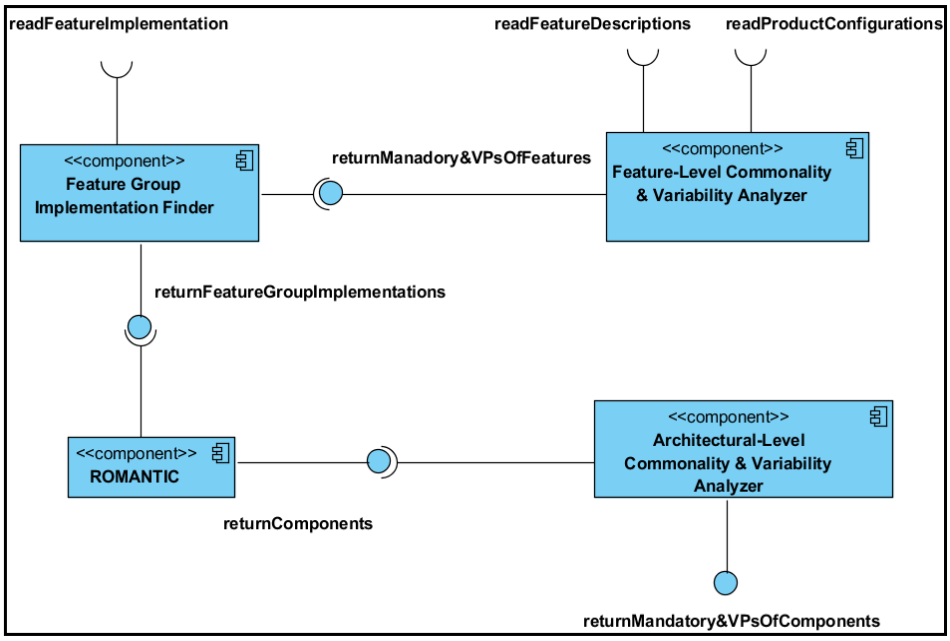

Figure A.4 : SPLA Implementation Architecture.

Table A.3 : Statistical source code information of Reverse Engineering SPLA Approach.

Products	**NOP**	**NOC**	**LOC**
Reverse Engineering SPLA	20	56	5,269

using the ROMANTIC. In this component, we link each feature to the corresponding component(s). We also conduct textual matching between component names for all features of each VP to find parent components. This component provides mandatory components and VPs of components.

Table A.3 presents statistical source code information of the reverse engineering SPLA approach in terms of number of packages (NOP), number of classes (NOC) and number of line of codes (LOC).

APPENDIX B

Product Configurations Used for Evaluation

In chapter *Toward Reverse Engineering SPLA from Product Variants: based on Feature Location*, we generated different sets of product configurations for ArgoUML-SPL, MobileMedia and MobilePhone. Below we present these sets for each case study as follows:

B.1 ArgoUML-SPL

Table B.1: Set 1 of product configurations for ArgoUML-SPL (256 Configuration)

Product	Features
Product1	Class
Product2	Class,State
Product3	Class,Activity
Product4	Class,State,Activity
Product5	Class,UseCase
Product6	Class,State,UseCase
Product7	Class,Activity,UseCase
Product8	Class,State,Activity,UseCase
Product9	Class,Collaboration
Product10	Class,State,Collaboration

Continued on next page

Table B.1 – *Continued from previous page*

Product	Features
Product11	Class,Activity,Collaboration
Product12	Class,State,Activity,Collaboration
Product13	Class,UseCase,Collaboration
Product14	Class,State,UseCase,Collaboration
Product15	Class,Activity,UseCase,Collaboration
Product16	Class,State,Activity,UseCase,Collaboration
Product17	Class,Deployment
Product18	Class,State,Deployment
Product19	Class,Activity,Deployment
Product20	Class,State,Activity,Deployment
Product21	Class,UseCase,Deployment
Product22	Class,State,UseCase,Deployment
Product23	Class,Activity,UseCase,Deployment
Product24	Class,State,Activity,UseCase,Deployment
Product25	Class,Collaboration,Deployment
Product26	Class,State,Collaboration,Deployment
Product27	Class,Activity,Collaboration,Deployment
Product28	Class,State,Activity,Collaboration,Deployment
Product29	Class,UseCase,Collaboration,Deployment
Product30	Class,State,UseCase,Collaboration,Deployment
Product31	Class,Activity,UseCase,Collaboration,Deployment
Product32	Class,State,Activity,UseCase,Collaboration,Deployment
Product33	Class,Sequence
Product34	Class,State,Sequence
Product35	Class,Activity,Sequence
Product36	Class,State,Activity,Sequence
Product37	Class,UseCase,Sequence
Product38	Class,State,UseCase,Sequence
Product39	Class,Activity,UseCase,Sequence
Product40	Class,State,Activity,UseCase,Sequence
Product41	Class,Collaboration,Sequence
Product42	Class,State,Collaboration,Sequence
Product43	Class,Activity,Collaboration,Sequence
Product44	Class,State,Activity,Collaboration,Sequence
Product45	Class,UseCase,Collaboration,Sequence
Product46	Class,State,UseCase,Collaboration,Sequence
Product47	Class,Activity,UseCase,Collaboration,Sequence
Product48	Class,State,Activity,UseCase,Collaboration,Sequence
Product49	Class,Deployment,Sequence
Product50	Class,State,Deployment,Sequence

Continued on next page

Table B.1 – *Continued from previous page*

Product	Features
Product51	Class,Activity,Deployment,Sequence
Product52	Class,State,Activity,Deployment,Sequence
Product53	Class,UseCase,Deployment,Sequence
Product54	Class,State,UseCase,Deployment,Sequence
Product55	Class,Activity,UseCase,Deployment,Sequence
Product56	Class,State,Activity,UseCase,Deployment,Sequence
Product57	Class,Collaboration,Deployment,Sequence
Product58	Class,State,Collaboration,Deployment,Sequence
Product59	Class,Activity,Collaboration,Deployment,Sequence
Product60	Class,State,Activity,Collaboration,Deployment,Sequence
Product61	Class,UseCase,Collaboration,Deployment,Sequence
Product62	Class,State,UseCase,Collaboration,Deployment,Sequence
Product63	Class,Activity,UseCase,Collaboration,Deployment,Sequence
Product64	Class,State,Activity,UseCase,Collaboration,Deployment,Sequence
Product65	Class,Cognitive
Product66	Class,State,Cognitive
Product67	Class,Activity,Cognitive
Product68	Class,State,Activity,Cognitive
Product69	Class,UseCase,Cognitive
Product70	Class,State,UseCase,Cognitive
Product71	Class,Activity,UseCase,Cognitive
Product72	Class,State,Activity,UseCase,Cognitive
Product73	Class,Collaboration,Cognitive
Product74	Class,State,Collaboration,Cognitive
Product75	Class,Activity,Collaboration,Cognitive
Product76	Class,State,Activity,Collaboration,Cognitive
Product77	Class,UseCase,Collaboration,Cognitive
Product78	Class,State,UseCase,Collaboration,Cognitive
Product79	Class,Activity,UseCase,Collaboration,Cognitive
Product80	Class,State,Activity,UseCase,Collaboration,Cognitive
Product81	Class,Deployment,Cognitive
Product82	Class,State,Deployment,Cognitive
Product83	Class,Activity,Deployment,Cognitive
Product84	Class,State,Activity,Deployment,Cognitive
Product85	Class,UseCase,Deployment,Cognitive
Product86	Class,State,UseCase,Deployment,Cognitive
Product87	Class,Activity,UseCase,Deployment,Cognitive
Product88	Class,State,Activity,UseCase,Deployment,Cognitive
Product89	Class,Collaboration,Deployment,Cognitive
Product90	Class,State,Collaboration,Deployment,Cognitive

Continued on next page

Table B.1 – *Continued from previous page*

Product	Features
Product91	Class,Activity,Collaboration,Deployment,Cognitive
Product92	Class,State,Activity,Collaboration,Deployment,Cognitive
Product93	Class,UseCase,Collaboration,Deployment,Cognitive
Product94	Class,State,UseCase,Collaboration,Deployment,Cognitive
Product95	Class,Activity,UseCase,Collaboration,Deployment,Cognitive
Product96	Class,State,Activity,UseCase,Collaboration,Deployment,Cognitive
Product97	Class,Sequence,Cognitive
Product98	Class,State,Sequence,Cognitive
Product99	Class,Activity,Sequence,Cognitive
Product100	Class,State,Activity,Sequence,Cognitive
Product101	Class,UseCase,Sequence,Cognitive
Product102	Class,State,UseCase,Sequence,Cognitive
Product103	Class,Activity,UseCase,Sequence,Cognitive
Product104	Class,State,Activity,UseCase,Sequence,Cognitive
Product105	Class,Collaboration,Sequence,Cognitive
Product106	Class,State,Collaboration,Sequence,Cognitive
Product107	Class,Activity,Collaboration,Sequence,Cognitive
Product108	Class,State,Activity,Collaboration,Sequence,Cognitive
Product109	Class,UseCase,Collaboration,Sequence,Cognitive
Product110	Class,State,UseCase,Collaboration,Sequence,Cognitive
Product111	Class,Activity,UseCase,Collaboration,Sequence,Cognitive
Product112	Class,State,Activity,UseCase,Collaboration,Sequence,Cognitive
Product113	Class,Deployment,Sequence,Cognitive
Product114	Class,State,Deployment,Sequence,Cognitive
Product115	Class,Activity,Deployment,Sequence,Cognitive
Product116	Class,State,Activity,Deployment,Sequence,Cognitive
Product117	Class,UseCase,Deployment,Sequence,Cognitive
Product118	Class,State,UseCase,Deployment,Sequence,Cognitive
Product119	Class,Activity,UseCase,Deployment,Sequence,Cognitive
Product120	Class,State,Activity,UseCase,Deployment,Sequence,Cognitive
Product121	Class,Collaboration,Deployment,Sequence,Cognitive
Product122	Class,State,Collaboration,Deployment,Sequence,Cognitive
Product123	Class,Activity,Collaboration,Deployment,Sequence,Cognitive
Product124	Class,State,Activity,Collaboration,Deployment,Sequence,Cognitive
Product125	Class,UseCase,Collaboration,Deployment,Sequence,Cognitive
Product126	Class,State,UseCase,Collaboration,Deployment,Sequence,Cognitive
Product127	Class,Activity,UseCase,Collaboration,Deployment,Sequence,Cognitive
Product128	Class,State,Activity,UseCase,Collaboration,Deployment,Sequence,Cognitive
Product129	Class,Logging
Product130	Class,State,Logging

Continued on next page

Table B.1 – *Continued from previous page*

Product	Features
Product131	Class,Activity,Logging
Product132	Class,State,Activity,Logging
Product133	Class,UseCase,Logging
Product134	Class,State,UseCase,Logging
Product135	Class,Activity,UseCase,Logging
Product136	Class,State,Activity,UseCase,Logging
Product137	Class,Collaboration,Logging
Product138	Class,State,Collaboration,Logging
Product139	Class,Activity,Collaboration,Logging
Product140	Class,State,Activity,Collaboration,Logging
Product141	Class,UseCase,Collaboration,Logging
Product142	Class,State,UseCase,Collaboration,Logging
Product143	Class,Activity,UseCase,Collaboration,Logging
Product144	Class,State,Activity,UseCase,Collaboration,Logging
Product145	Class,Deployment,Logging
Product146	Class,State,Deployment,Logging
Product147	Class,Activity,Deployment,Logging
Product148	Class,State,Activity,Deployment,Logging
Product149	Class,UseCase,Deployment,Logging
Product150	Class,State,UseCase,Deployment,Logging
Product151	Class,Activity,UseCase,Deployment,Logging
Product152	Class,State,Activity,UseCase,Deployment,Logging
Product153	Class,Collaboration,Deployment,Logging
Product154	Class,State,Collaboration,Deployment,Logging
Product155	Class,Activity,Collaboration,Deployment,Logging
Product156	Class,State,Activity,Collaboration,Deployment,Logging
Product157	Class,UseCase,Collaboration,Deployment,Logging
Product158	Class,State,UseCase,Collaboration,Deployment,Logging
Product159	Class,Activity,UseCase,Collaboration,Deployment,Logging
Product160	Class,State,Activity,UseCase,Collaboration,Deployment,Logging
Product161	Class,Sequence,Logging
Product162	Class,State,Sequence,Logging
Product163	Class,Activity,Sequence,Logging
Product164	Class,State,Activity,Sequence,Logging
Product165	Class,UseCase,Sequence,Logging
Product166	Class,State,UseCase,Sequence,Logging
Product167	Class,Activity,UseCase,Sequence,Logging
Product168	Class,State,Activity,UseCase,Sequence,Logging
Product169	Class,Collaboration,Sequence,Logging
Product170	Class,State,Collaboration,Sequence,Logging

Continued on next page

Table B.1 – *Continued from previous page*

Product	Features
Product171	Class,Activity,Collaboration,Sequence,Logging
Product172	Class,State,Activity,Collaboration,Sequence,Logging
Product173	Class,UseCase,Collaboration,Sequence,Logging
Product174	Class,State,UseCase,Collaboration,Sequence,Logging
Product175	Class,Activity,UseCase,Collaboration,Sequence,Logging
Product176	Class,State,Activity,UseCase,Collaboration,Sequence,Logging
Product177	Class,Deployment,Sequence,Logging
Product178	Class,State,Deployment,Sequence,Logging
Product179	Class,Activity,Deployment,Sequence,Logging
Product180	Class,State,Activity,Deployment,Sequence,Logging
Product181	Class,UseCase,Deployment,Sequence,Logging
Product182	Class,State,UseCase,Deployment,Sequence,Logging
Product183	Class,Activity,UseCase,Deployment,Sequence,Logging
Product184	Class,State,Activity,UseCase,Deployment,Sequence,Logging
Product185	Class,Collaboration,Deployment,Sequence,Logging
Product186	Class,State,Collaboration,Deployment,Sequence,Logging
Product187	Class,Activity,Collaboration,Deployment,Sequence,Logging
Product188	Class,State,Activity,Collaboration,Deployment,Sequence,Logging
Product189	Class,UseCase,Collaboration,Deployment,Sequence,Logging
Product190	Class,State,UseCase,Collaboration,Deployment,Sequence,Logging
Product191	Class,Activity,UseCase,Collaboration,Deployment,Sequence,Logging
Product192	Class,State,Activity,UseCase,Collaboration,Deployment,Sequence,Logging
Product193	Class,Cognitive,Logging
Product194	Class,State,Cognitive,Logging
Product195	Class,Activity,Cognitive,Logging
Product196	Class,State,Activity,Cognitive,Logging
Product197	Class,UseCase,Cognitive,Logging
Product198	Class,State,UseCase,Cognitive,Logging
Product199	Class,Activity,UseCase,Cognitive,Logging
Product200	Class,State,Activity,UseCase,Cognitive,Logging
Product201	Class,Collaboration,Cognitive,Logging
Product202	Class,State,Collaboration,Cognitive,Logging
Product203	Class,Activity,Collaboration,Cognitive,Logging
Product204	Class,State,Activity,Collaboration,Cognitive,Logging
Product205	Class,UseCase,Collaboration,Cognitive,Logging
Product206	Class,State,UseCase,Collaboration,Cognitive,Logging
Product207	Class,Activity,UseCase,Collaboration,Cognitive,Logging
Product208	Class,State,Activity,UseCase,Collaboration,Cognitive,Logging
Product209	Class,Deployment,Cognitive,Logging
Product210	Class,State,Deployment,Cognitive,Logging

Continued on next page

Table B.1 – *Continued from previous page*

Product	Features
Product211	Class,Activity,Deployment,Cognitive,Logging
Product212	Class,State,Activity,Deployment,Cognitive,Logging
Product213	Class,UseCase,Deployment,Cognitive,Logging
Product214	Class,State,UseCase,Deployment,Cognitive,Logging
Product215	Class,Activity,UseCase,Deployment,Cognitive,Logging
Product216	Class,State,Activity,UseCase,Deployment,Cognitive,Logging
Product217	Class,Collaboration,Deployment,Cognitive,Logging
Product218	Class,State,Collaboration,Deployment,Cognitive,Logging
Product219	Class,Activity,Collaboration,Deployment,Cognitive,Logging
Product220	Class,State,Activity,Collaboration,Deployment,Cognitive,Logging
Product221	Class,UseCase,Collaboration,Deployment,Cognitive,Logging
Product222	Class,State,UseCase,Collaboration,Deployment,Cognitive,Logging
Product223	Class,Activity,UseCase,Collaboration,Deployment,Cognitive,Logging
Product224	Class,State,Activity,UseCase,Collaboration,Deployment,Cognitive,Logging
Product225	Class,Sequence,Cognitive,Logging
Product226	Class,State,Sequence,Cognitive,Logging
Product227	Class,Activity,Sequence,Cognitive,Logging
Product228	Class,State,Activity,Sequence,Cognitive,Logging
Product229	Class,UseCase,Sequence,Cognitive,Logging
Product230	Class,State,UseCase,Sequence,Cognitive,Logging
Product231	Class,Activity,UseCase,Sequence,Cognitive,Logging
Product232	Class,State,Activity,UseCase,Sequence,Cognitive,Logging
Product233	Class,Collaboration,Sequence,Cognitive,Logging
Product234	Class,State,Collaboration,Sequence,Cognitive,Logging
Product235	Class,Activity,Collaboration,Sequence,Cognitive,Logging
Product236	Class,State,Activity,Collaboration,Sequence,Cognitive,Logging
Product237	Class,UseCase,Collaboration,Sequence,Cognitive,Logging
Product238	Class,State,UseCase,Collaboration,Sequence,Cognitive,Logging
Product239	Class,Activity,UseCase,Collaboration,Sequence,Cognitive,Logging
Product240	Class,State,Activity,UseCase,Collaboration,Sequence,Cognitive,Logging
Product241	Class,Deployment,Sequence,Cognitive,Logging
Product242	Class,State,Deployment,Sequence,Cognitive,Logging
Product243	Class,Activity,Deployment,Sequence,Cognitive,Logging
Product244	Class,State,Activity,Deployment,Sequence,Cognitive,Logging
Product245	Class,UseCase,Deployment,Sequence,Cognitive,Logging
Product246	Class,State,UseCase,Deployment,Sequence,Cognitive,Logging
Product247	Class,Activity,UseCase,Deployment,Sequence,Cognitive,Logging
Product248	Class,State,Activity,UseCase,Deployment,Sequence,Cognitive,Logging
Product249	Class,Collaboration,Deployment,Sequence,Cognitive,Logging
Product250	Class,State,Collaboration,Deployment,Sequence,Cognitive,Logging

Continued on next page

Table B.1 – *Continued from previous page*

Product	Features
Product251	Class,Activity,Collaboration,Deployment,Sequence,Cognitive,Logging
Product252	Class,State,Activity,Collaboration,Deployment,Sequence,Cognitive,Logging
Product253	Class,UseCase,Collaboration,Deployment,Sequence,Cognitive,Logging
Product254	Class,State,UseCase,Collaboration,Deployment,Sequence,Cognitive,Logging
Product255	Class,Activity,UseCase,Collaboration,Deployment,Sequence,Cognitive,Logging
Product256	Class,State,Activity,UseCase,Collaboration,Deployment,Sequence,Cognitive,Logging

Table B.2 : Set 2 of product configurations for ArgoUML-SPL (7 Configuration)

Product	Features
Product1	Class, Cognitive, Sequence, Collaboration, Deployment, Usecase, Activity, State, Logging
Product2	Class, Cognitive, Sequence, Collaboration, Deployment, Usecase.
Product3	Class, Cognitive, Sequence, Activity, State, Usecase.
Product4	Class, Cognitive, Activity, Collaboration, Deployment, State.
Product5	Class, Cognitive.
Product6	Class, Cognitive, Sequence, Usecase.
Product7	Class, Cognitive, Collaboration, Deployment.

B.2 MobileMedia

Table B.3 : Set 1 of product configurations for MobileMedia (16 Configuration)

Product	Features
Product1	Album_Management,Photo_Management
Product2	Album_Management,Photo_Management,Edit_Photo_Label
Product3	Album_Management,Photo_Management,Copy_Photo
Product4	Album_Management,Photo_Management,Edit_Photo_Label,Copy_Photo
Product5	Album_Management,Photo_Management,SMS_Transfer
Product6	Album_Management,Photo_Management,Edit_Photo_Label,SMS_Transfer
Product7	Album_Management,Photo_Management,Copy_Photo,SMS_Transfer
Product8	Album_Management,Photo_Management,Edit_Photo_Label,Copy_Photo,SMS_Transfer
Product9	Album_Management,Photo_Management,Exception_Handling
Product10	Album_Management,Photo_Management,Edit_Photo_Label,Exception_Handling
Product11	Album_Management,Photo_Management,Copy_Photo,Exception_Handling
Product12	Album_Management,Photo_Management,Edit_Photo_Label,Copy_Photo,Exception_Handling
Product13	Album_Management,Photo_Management,SMS_Transfer,Exception_Handling
Product14	Album_Management,Photo_Management,Edit_Photo_Label,SMS_Transfer,Exception_Handling
Product15	Album_Management,Photo_Management,Copy_Photo,SMS_Transfer,Exception_Handling
Product16	Album_Management,Photo_Management,Edit_Photo_Label,Copy_Photo,SMS_Transfer,Exception_Handling

Table B.4 : Set 2 of product configurations for MobileMedia (8 Configuration)

Product	Features
Product1	Album_Management,Photo_Management
Product2	Album_Management,Photo_Management,Edit_Photo_Label
Product3	Album_Management,Photo_Management,Copy_Photo
Product4	Album_Management,Photo_Management,Edit_Photo_Label,Copy_Photo
Product5	Album_Management,Photo_Management,SMS_Transfer
Product6	Album_Management,Photo_Management,Edit_Photo_Label,Exception_Handling
Product7	Album_Management,Photo_Management,Exception_Handling
Product8	Album_Management,Photo_Management,SMS_Transfer,Exception_Handling

Table B.5 : Set 3 of product configurations for MobileMedia (5 Configuration)

Product	Features
Product1	Album_Management, Photo_Management
Product2	Exception_Handling, Album_Management, Photo_Management
Product3	Exception_Handling, Edit_Photo_Label, Album_Management, Photo_Management
Product4	Exception_Handling, Copy_Photo, Edit_Photo_Label, Album_Management, Photo_Management
Product5	Exception_Handling, SMS_Transfer, Copy_Photo, Edit_Photo_Label, Album_Management, Photo_Management

B.3 MobilePhone

Table B.6: Set 1 of product configurations for MobilePhone (224 Configuration)

Product	Features
Product1	Basic, Call, SMS, Android
Product2	Basic, Call, SMS, IOS
Product3	HighResolution, Call, SMS, Android
Product4	HighResolution, Call, SMS, IOS
Product5	Touch, Call, SMS, Android
Product6	Touch, Call, SMS, IOS
Product7	GPS, HighResolution, Call, SMS, Android
Product8	GPS, HighResolution, Call, SMS, IOS
Product9	GPS, Touch, Call, SMS, Android
Product10	GPS, Touch, Call, SMS, IOS
Product11	HighResolution, Call, Camera, SMS, Android
Product12	HighResolution, Call, Camera, SMS, IOS
Product13	Basic, Call, MP3, SMS, Android
Product14	Basic, Call, MP3, SMS, IOS
Product15	HighResolution, Call, MP3, SMS, Android
Product16	HighResolution, Call, MP3, SMS, IOS
Product17	Touch, Call, MP3, SMS, Android
Product18	Touch, Call, MP3, SMS, IOS
Product19	HighResolution, Call, Camera, MP3, SMS, Android
Product20	HighResolution, Call, Camera, MP3, SMS, IOS
Product21	Basic, Call, Radio, SMS, Android
Product22	Basic, Call, Radio, SMS, IOS
Product23	HighResolution, Call, Radio, SMS, Android

Continued on next page

Table B.6 – *Continued from previous page*

Product	Features
Product24	HighResolution, Call, Radio, SMS, IOS
Product25	Touch, Call, Radio, SMS, Android
Product26	Touch, Call, Radio, SMS, IOS
Product27	HighResolution, Call, Camera, Radio, SMS, Android
Product28	HighResolution, Call, Camera, Radio, SMS, IOS
Product29	Basic, Call, MP3, Radio, SMS, Android
Product30	Basic, Call, MP3, Radio, SMS, IOS
Product31	HighResolution, Call, MP3, Radio, SMS, Android
Product32	HighResolution, Call, MP3, Radio, SMS, IOS
Product33	Touch, Call, MP3, Radio, SMS, Android
Product34	Touch, Call, MP3, Radio, SMS, IOS
Product35	HighResolution, Call, Camera, MP3, Radio, SMS, Android
Product36	HighResolution, Call, Camera, MP3, Radio, SMS, IOS
Product37	GPS, HighResolution, Call, Camera, SMS, Android
Product38	GPS, HighResolution, Call, Camera, SMS, IOS
Product39	GPS, HighResolution, Call, MP3, SMS, Android
Product40	GPS, HighResolution, Call, MP3, SMS, IOS
Product41	GPS, Touch, Call, MP3, SMS, Android
Product42	GPS, Touch, Call, MP3, SMS, IOS
Product43	GPS, HighResolution, Call, Camera, MP3, SMS, Android
Product44	GPS, HighResolution, Call, Camera, MP3, SMS, IOS
Product45	GPS, HighResolution, Call, Radio, SMS, Android
Product46	GPS, HighResolution, Call, Radio, SMS, IOS
Product47	GPS, Touch, Call, Radio, SMS, Android
Product48	GPS, Touch, Call, Radio, SMS, IOS
Product49	GPS, HighResolution, Call, Camera, Radio, SMS, Android
Product50	GPS, HighResolution, Call, Camera, Radio, SMS, IOS
Product51	GPS, HighResolution, Call, MP3, Radio, SMS, Android
Product52	GPS, HighResolution, Call, MP3, Radio, SMS, IOS
Product53	GPS, Touch, Call, MP3, Radio, SMS, Android
Product54	GPS, Touch, Call, MP3, Radio, SMS, IOS
Product55	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, Android
Product56	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, IOS
Product57	Basic, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product58	Basic, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product59	HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product60	HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product61	Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product62	Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS

Continued on next page

Table B.6 – *Continued from previous page*

Product	**Features**
Product63	GPS, HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product64	GPS, HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product65	GPS, Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product66	GPS, Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product67	HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product68	HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product69	Basic, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product70	Basic, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product71	HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product72	HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product73	Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product74	Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product75	HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product76	HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product77	Basic, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product78	Basic, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product79	HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product80	HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product81	Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product82	Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product83	HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product84	HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product85	Basic, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product86	Basic, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product87	HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product88	HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product89	Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product90	Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS

Continued on next page

Table B.6 – *Continued from previous page*

Product	Features
Product91	HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product92	HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product93	GPS, HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product94	GPS, HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product95	GPS, HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product96	GPS, HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product97	GPS, Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product98	GPS, Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product99	GPS, HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product100	GPS, HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product101	GPS, HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product102	GPS, HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product103	GPS, Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product104	GPS, Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product105	GPS, HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product106	GPS, HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product107	GPS, HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product108	GPS, HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product109	GPS, Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product110	GPS, Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS
Product111	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product112	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, IOS

Continued on next page

Table B.6 – *Continued from previous page*

Product	Features
Product113	Basic, Call, SMS, Recorder, Android
Product114	Basic, Call, SMS, Recorder, IOS
Product115	HighResolution, Call, SMS, Recorder, Android
Product116	HighResolution, Call, SMS, Recorder, IOS
Product117	Touch, Call, SMS, Recorder, Android
Product118	Touch, Call, SMS, Recorder, IOS
Product119	GPS, HighResolution, Call, SMS, Recorder, Android
Product120	GPS, HighResolution, Call, SMS, Recorder, IOS
Product121	GPS, Touch, Call, SMS, Recorder, Android
Product122	GPS, Touch, Call, SMS, Recorder, IOS
Product123	HighResolution, Call, Camera, SMS, Recorder, Android
Product124	HighResolution, Call, Camera, SMS, Recorder, IOS
Product125	Basic, Call, MP3, SMS, Recorder, Android
Product126	Basic, Call, MP3, SMS, Recorder, IOS
Product127	HighResolution, Call, MP3, SMS, Recorder, Android
Product128	HighResolution, Call, MP3, SMS, Recorder, IOS
Product129	Touch, Call, MP3, SMS, Recorder, Android
Product130	Touch, Call, MP3, SMS, Recorder, IOS
Product131	HighResolution, Call, Camera, MP3, SMS, Recorder, Android
Product132	HighResolution, Call, Camera, MP3, SMS, Recorder, IOS
Product133	Basic, Call, Radio, SMS, Recorder, Android
Product134	Basic, Call, Radio, SMS, Recorder, IOS
Product135	HighResolution, Call, Radio, SMS, Recorder, Android
Product136	HighResolution, Call, Radio, SMS, Recorder, IOS
Product137	Touch, Call, Radio, SMS, Recorder, Android
Product138	Touch, Call, Radio, SMS, Recorder, IOS
Product139	HighResolution, Call, Camera, Radio, SMS, Recorder, Android
Product140	HighResolution, Call, Camera, Radio, SMS, Recorder, IOS
Product141	Basic, Call, MP3, Radio, SMS, Recorder, Android
Product142	Basic, Call, MP3, Radio, SMS, Recorder, IOS
Product143	HighResolution, Call, MP3, Radio, SMS, Recorder, Android
Product144	HighResolution, Call, MP3, Radio, SMS, Recorder, IOS
Product145	Touch, Call, MP3, Radio, SMS, Recorder, Android
Product146	Touch, Call, MP3, Radio, SMS, Recorder, IOS
Product147	HighResolution, Call, Camera, MP3, Radio, SMS, Recorder, Android
Product148	HighResolution, Call, Camera, MP3, Radio, SMS, Recorder, IOS
Product149	GPS, HighResolution, Call, Camera, SMS, Recorder, Android
Product150	GPS, HighResolution, Call, Camera, SMS, Recorder, IOS
Product151	GPS, HighResolution, Call, MP3, SMS, Recorder, Android
Product152	GPS, HighResolution, Call, MP3, SMS, Recorder, IOS

Continued on next page

Table B.6 – *Continued from previous page*

Product	Features
Product153	GPS, Touch, Call, MP3, SMS, Recorder, Android
Product154	GPS, Touch, Call, MP3, SMS, Recorder, IOS
Product155	GPS, HighResolution, Call, Camera, MP3, SMS, Recorder, Android
Product156	GPS, HighResolution, Call, Camera, MP3, SMS, Recorder, IOS
Product157	GPS, HighResolution, Call, Radio, SMS, Recorder, Android
Product158	GPS, HighResolution, Call, Radio, SMS, Recorder, IOS
Product159	GPS, Touch, Call, Radio, SMS, Recorder, Android
Product160	GPS, Touch, Call, Radio, SMS, Recorder, IOS
Product161	GPS, HighResolution, Call, Camera, Radio, SMS, Recorder, Android
Product162	GPS, HighResolution, Call, Camera, Radio, SMS, Recorder, IOS
Product163	GPS, HighResolution, Call, MP3, Radio, SMS, Recorder, Android
Product164	GPS, HighResolution, Call, MP3, Radio, SMS, Recorder, IOS
Product165	GPS, Touch, Call, MP3, Radio, SMS, Recorder, Android
Product166	GPS, Touch, Call, MP3, Radio, SMS, Recorder, IOS
Product167	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, Recorder, Android
Product168	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, Recorder, IOS
Product169	Basic, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product170	Basic, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product171	HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product172	HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product173	Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product174	Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product175	GPS, HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product176	GPS, HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product177	GPS, Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product178	GPS, Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product179	HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product180	HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product181	Basic, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product182	Basic, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product183	HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android

Continued on next page

Table B.6 – *Continued from previous page*

Product	Features
Product184	HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product185	Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product186	Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product187	HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product188	HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product189	Basic, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product190	Basic, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product191	HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product192	HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product193	Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product194	Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product195	HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product196	HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product197	Basic, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product198	Basic, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product199	HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product200	HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product201	Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product202	Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product203	HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product204	HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS

Continued on next page

Table B.6 – *Continued from previous page*

Product	Features
Product205	GPS, HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product206	GPS, HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product207	GPS, HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product208	GPS, HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product209	GPS, Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product210	GPS, Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product211	GPS, HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product212	GPS, HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product213	GPS, HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product214	GPS, HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product215	GPS, Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product216	GPS, Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product217	GPS, HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product218	GPS, HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product219	GPS, HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product220	GPS, HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product221	GPS, Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product222	GPS, Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product223	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product224	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS

Table B.7: Set 2 of product configurations for MobilePhone(112 Configuration)

Product	Features
Product1	HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product2	Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product3	HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product4	GPS, HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product5	GPS, HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product6	GPS, Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product7	GPS, Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS
Product8	Basic, Call, SMS, Android
Product9	HighResolution, Call, SMS, Android
Product10	Touch, Call, SMS, Android
Product11	GPS, HighResolution, Call, SMS, Android
Product12	GPS, Touch, Call, SMS, Android
Product13	HighResolution, Call, Camera, SMS, Android
Product14	Basic, Call, MP3, SMS, Android
Product15	HighResolution, Call, MP3, SMS, Android
Product16	GPS, HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product17	GPS, Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product18	GPS, HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product19	GPS, HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product20	GPS, Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product21	GPS, HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product22	Touch, Call, MP3, SMS, Android
Product23	HighResolution, Call, Camera, MP3, SMS, Android

Continued on next page

Table B.7 – *Continued from previous page*

Product	Features
Product24	Basic, Call, Radio, SMS, Android
Product25	HighResolution, Call, Radio, SMS, Android
Product26	Touch, Call, Radio, SMS, Android
Product27	HighResolution, Call, Camera, Radio, SMS, Android
Product28	Basic, Call, MP3, Radio, SMS, Android
Product29	HighResolution, Call, MP3, Radio, SMS, Android
Product30	Touch, Call, MP3, Radio, SMS, Android
Product31	GPS, HighResolution, Call, Camera, SMS, Android
Product32	GPS, HighResolution, Call, MP3, SMS, Android
Product33	GPS, Touch, Call, MP3, SMS, Android
Product34	GPS, HighResolution, Call, Camera, MP3, SMS, Android
Product35	GPS, HighResolution, Call, Radio, SMS, Android
Product36	HighResolution, Call, Camera, MP3, Radio, SMS, Android
Product37	GPS, Touch, Call, Radio, SMS, Android
Product38	GPS, HighResolution, Call, MP3, Radio, SMS, Android
Product39	GPS, Touch, Call, MP3, Radio, SMS, Android
Product40	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, Android
Product41	Basic, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product42	HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product43	Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product44	GPS, HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product45	GPS, HighResolution, Call, Camera, Radio, SMS, Android
Product46	GPS, Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product47	HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product48	Basic, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product49	Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product50	HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product51	HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product52	Basic, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product53	HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product54	HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product55	Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product56	Basic, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android

Continued on next page

Table B.7 – *Continued from previous page*

Product	Features
Product57	HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product58	HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product59	Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product60	GPS, HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product61	GPS, HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product62	GPS, Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product63	GPS, HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product64	GPS, HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product65	GPS, Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product66	GPS, HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product67	GPS, Touch, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product68	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product69	Basic, Call, SMS, Recorder, Android
Product70	GPS, HighResolution, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product71	HighResolution, Call, SMS, Recorder, Android
Product72	Touch, Call, SMS, Recorder, Android
Product73	GPS, HighResolution, Call, SMS, Recorder, Android
Product74	GPS, Touch, Call, SMS, Recorder, Android
Product75	Basic, Call, MP3, SMS, Recorder, Android
Product76	HighResolution, Call, MP3, SMS, Recorder, Android
Product77	Touch, Call, MP3, SMS, Recorder, Android
Product78	HighResolution, Call, Camera, SMS, Recorder, Android
Product79	HighResolution, Call, Camera, MP3, SMS, Recorder, Android
Product80	Basic, Call, Radio, SMS, Recorder, Android
Product81	HighResolution, Call, Radio, SMS, Recorder, Android
Product82	Touch, Call, Radio, SMS, Recorder, Android
Product83	HighResolution, Call, Camera, Radio, SMS, Recorder, Android
Product84	Basic, Call, MP3, Radio, SMS, Recorder, Android
Product85	HighResolution, Call, MP3, Radio, SMS, Recorder, Android
Product86	Touch, Call, MP3, Radio, SMS, Recorder, Android

Continued on next page

Table B.7 – *Continued from previous page*

Product	Features
Product87	HighResolution, Call, Camera, MP3, Radio, SMS, Recorder, Android
Product88	GPS, HighResolution, Call, Camera, SMS, Recorder, Android
Product89	GPS, HighResolution, Call, MP3, SMS, Recorder, Android
Product90	GPS, Touch, Call, MP3, SMS, Recorder, Android
Product91	GPS, HighResolution, Call, Camera, MP3, SMS, Recorder, Android
Product92	GPS, HighResolution, Call, Radio, SMS, Recorder, Android
Product93	GPS, Touch, Call, Radio, SMS, Recorder, Android
Product94	GPS, HighResolution, Call, Camera, Radio, SMS, Recorder, Android
Product95	GPS, HighResolution, Call, MP3, Radio, SMS, Recorder, Android
Product96	GPS, Touch, Call, MP3, Radio, SMS, Recorder, Android
Product97	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, Recorder, Android
Product98	Basic, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product99	HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product100	Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product101	GPS, HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product102	GPS, Touch, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product103	HighResolution, Call, Camera, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product104	Basic, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product105	HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product106	Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product107	HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product108	Basic, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product109	HighResolution, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product110	Touch, Call, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product111	HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product112	Basic, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android

Table B.8 : Set 3 of product configurations for MobilePhone (10 Configuration)

Product	Features
Product1	GPS, HighResolution, Call, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product2	Basic, Call, SMS, Android
Product3	Touch, Call ,Radio, SMS, Android
Product4	Basic, Call, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product5	GPS, HighResolution, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product6	HighResolution, Call, Camera, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Android
Product7	Basic, Call, MP3, SMS, Recorder, Android
Product8	Touch, Call, MP3, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product9	HighResolution, Call, Camera, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, Android
Product10	GPS, HighResolution, Call, Camera, MP3, Radio, SMS, SendPhoto, ReceivePhoto, PhotoExplorer, Recorder, IOS

APPENDIX C

Component Extraction from Object-Oriented Source Code: ROMANTIC Approach

ROMANTIC (**R**e-engineering of **O**bject-oriented syste**M**s by **A**rchitecture extractio**N** and migra**TI**on to **C**omponent based ones.) is an approach to automatically recover a component-based architecture from the source code of an existing object-oriented software [Kebir *et al.*, 2012b][Chardigny *et al.*, 2008]. ROMANTIC relies on two models: *mapping model* and *quality model*.

- A mapping model between object-oriented concepts (i.e. classes, interfaces, packages, etc.) and component-based software engineering ones (i.e. components, interfaces, sub-component, etc.).
- A measurement model of semantic-correctness of a component. This model refines characteristics of a component to measurable metrics. Based on these metrics, a fitness function is defined to measure the semantic-correctness of a component.

C.1 Mapping Model between Component and Object Concepts

In ROMANTIC, components are as disjoint collections of classes. Each collection is named *shape* and contains classes which can belong to different object-oriented packages (see Figure C.1). The classes of a shape are organized into two sets to constitute respectively shape *interface* and *center*. The shape interface classes have links with some classes from the outside of the shape using a method call or attribute use; while the remaining classes represent the shape center. Figure C.1 shows the

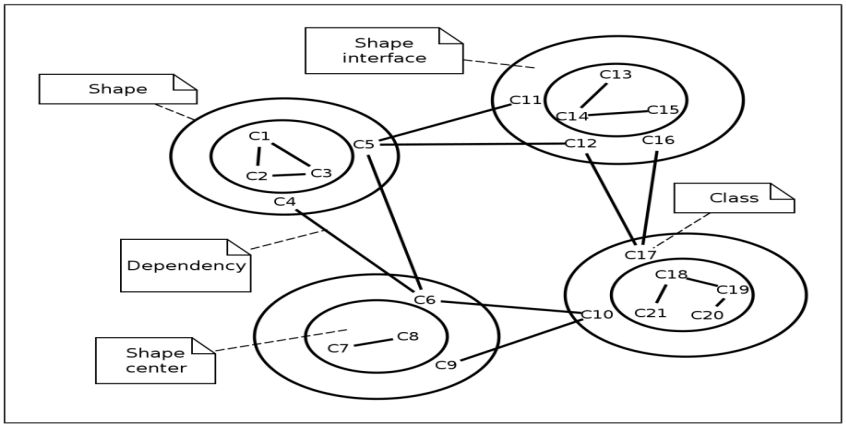

Figure C.1 : Shape Structure [Kebir *et al.*, 2012b][Chardigny *et al.*, 2008].

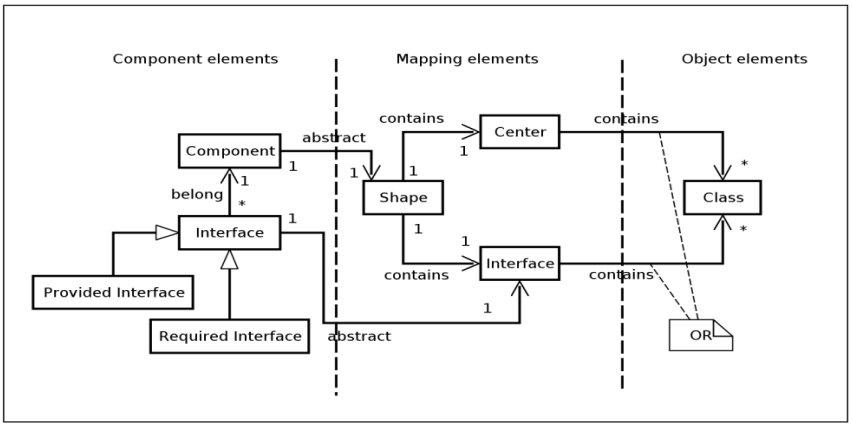

Figure C.2 : Object-Component Mapping Model [Kebir *et al.*, 2012b][Chardigny *et al.*, 2008].

shape (resp. component) structure. Figure C.2 shows the component-object mapping model that was proposed to handle the correspondence between object and component concepts.

C.2 Semantic-Correctness of Components

The semantic correctness of a component is based on the component characteristics. These characteristics are identified by studying the most commonly admitted definitions of software component. By combining and refining the common elements of different definitions of the component concept, three semantic characteristics of software components were identified: *composability*, *autonomy* and *specificity*.

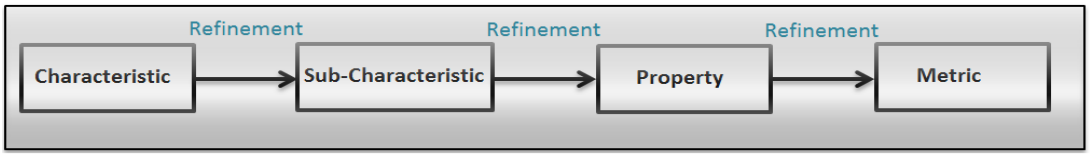

Figure C.3 : MetaModel of Refinement Software Characteristics Norm ISO-9126 [Kebir *et al.*, 2012b][Chardigny *et al.*, 2008].

C.2.1 From Characteristics to Properties

The identified semantic correctness characteristics are measured by using the refinement model given by the norm ISO-9126 (see Figure C.3) [ISO, 2001]. Based on this model, the semantic correctness of a component represents a characteristic and the components characteristics represent sub-characteristics. Also according to this model, these sub-characteristics are refined into measurable properties. This refinement is done using the semantic which is associated with these sub-characteristics. Below, we present such a refinement.

- **Autonomy**: A component is autonomous if it has no required interfaces, and hence the property number of required interfaces should lead to a good measure of the component autonomy.
- **Composability**: A component can be composed by means of its provided and required interfaces. A component was considered more easily composed with another if services provided/required in each interface are cohesive. The property average of service cohesion of component interface was used to measure composability.
- **Specificity**: The specificity characteristic of a component is refined to properties by the evaluation of the number of provided services, which are based on the following statements. Firstly, a component which provides many interfaces may provide various services, as an interface can offer different services. Thus the higher the number of interfaces is, the higher the number of provided services. Secondly, if interfaces (resp. services in each interface) are cohesive (i.e. share resources), they probably offer closely related functionalities. Thirdly, if the code of the component is loosely coupled (resp. cohesive), the different parts of the component code use each other (resp. common resources). Consequently, they probably work together in order to offer a few functionalities. From these statements, the specificity characteristic was refined to the following properties: number of provided interfaces, average of service cohesion of component interface, component interface cohesion, and component cohesion and coupling.

C.2.2 From Properties to Metrics

According to the refinement model given by the norm ISO-9126, a set of metrics are needed to measure the components properties mentioned above. In order to define these metrics, a link between component properties and shape properties is needed. Such a corresponding link is established as follows:

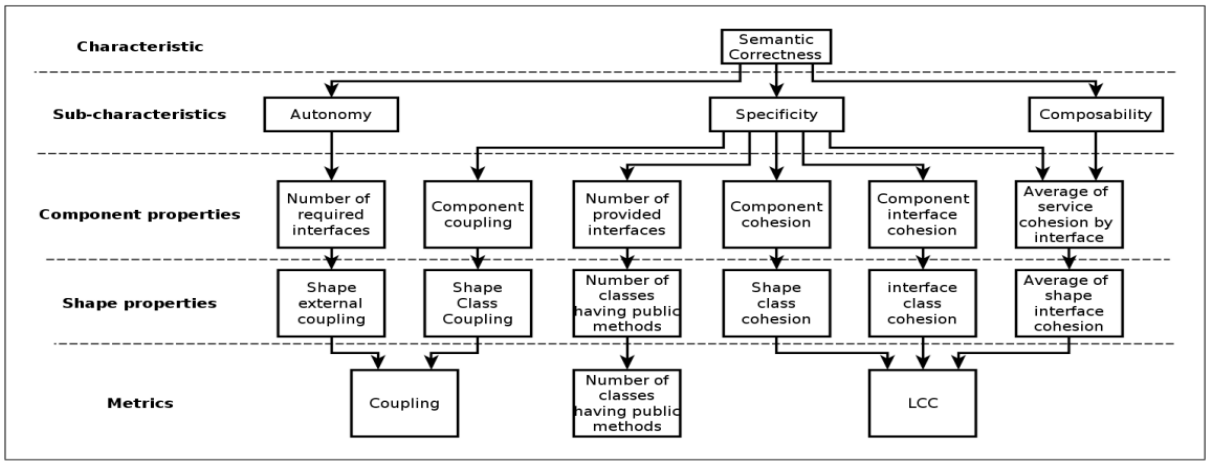

Figure C.4 : The Refinement Model for Semantic-Correctness [Kebir *et al.*, 2012b][Chardigny *et al.*, 2008].

- Firstly, according to the mapping model, component interface set is linked to the shape interface set. Thus, the average of the interface-class cohesion gives a correct measure for the property of the average of service cohesion of a component interface.
- Secondly, the component interface cohesion, the internal component cohesion and the internal component coupling can respectively be measured by the properties of interface class cohesion, shape class cohesion and shape class coupling.
- Thirdly, to link the property of the number of provided interfaces to a shape property, a component provided interface is associated to each shape-interface class having public methods. Thus, the number of provided interfaces is measured using the number of shape interface classes having public methods.
- Finally, the number of required interfaces is evaluated by using coupling between the component and the outside. This coupling is linked to shape external coupling. Thus, this property is measured using the property shape external coupling.

The properties *shape class coupling* and *shape external coupling* require a coupling measurement. Thus, the metric Coupl(E) and CouplExt(E) are defined to measure respectively the coupling of a shape E and the coupling of E with the outside classes. They measure two types of dependencies between objects: method calls and use of attributes of another class. Moreover, they are related through the equation below:

$$couplExt(E) = 100 - coupl(E). \tag{C.1}$$

The properties *average of interface-class cohesion*, *interface-class cohesion*, and *shape-class cohesion* require a cohesion measurement. The metric Loose Class Cohesion (LCC), proposed by Bieman and Kang [Bieman et Kang, 1995], was used to measure the percentage of pairs of methods which are directly or indirectly connected. The refinement model of the semantic correctness of a component is summarized in Figure C.4.

C.2.3 Evaluation of the Semantic-Correctness

According to the established mapping between the sub-characteristics, properties and metrics, three evaluation functions were proposed *Spe*, *A* and *C* respectively for *specificity*, *autonomy* and *composability*, where *nbPub(I)* is the number of interface classes having a public method and *I* is the shape interface of the shape *E*:

1. $Spe(E) = \frac{1}{5} \cdot (\frac{1}{|I|} \cdot \sum_{i \in I} LCC(i) + LCC(I) + LCC(E) + Coupl(E) + nbPub(I))$
2. $A(E) = couplExt(E) = 100 - coupl(E)$
3. $C(E) = \frac{1}{|I|} \cdot \sum_{i \in I} LCC(i)$

The function *S(E)* defined below represents the semantic-correctness function of a shape *E*. This function is based on the evaluation of each sub-characteristic. That is why it is as a linear combination of each fitness function of sub-characteristics (i.e. *Spe*, *A* and *C*):

4. $S(E) = \frac{1}{\sum_{i=1}^{3} \lambda_i} \cdot (\lambda_1 \cdot Spe(E) + \lambda_2 \cdot A(E) + \lambda_3 \cdot C(E))$

C.3 Naming Components

In ROMANTIC, naming component was performed based on the following observations: in many object-oriented languages, class names are a sequence of nouns concatenated using a camel-case notation (i.e. StringBuffer, ElementFilter, etc). The first word of a class name indicates the main purpose of the class; the second word indicate a complementary purpose of the class and so on. On the other hand, an interface name should be an adjective that qualifies its implementing class. According to these observations, three steps were proposed for naming components: extracting and tokenizing class and interface names from identified components, weighting tokens and constructing the component name.

C.3.1 Extracting and Tokenizing Class and Interface Names

In this step, class and interface names are extracted and then split into tokens according to the camel-case syntax. For example: *StringBuffer* is split into *String* and *Buffer*.

C.3.2 Weighting Tokens

In this step, a weight is assigned to each extracted token. A large weight is given to tokens that constitute the first word of class names. A medium weight is given to tokens that are the first word of interface names. Finally a small weight is given to the other tokens.

As a component consists of two sets of classes (center and interface), two strategies were proposed to deal with these classes for naming component purpose. A large weight was given to tokens extracted from classes that belong to the provided interface of a component because these classes constitute the provided functionalities and services that it offers to other components, thus, its main

purpose. A small weight was given to tokens extracted from classes that belong to the center of a component because these classes are less concerned with the main purpose of the component and are mainly utility classes that do not interact with the outside. For a given word (w), the weight is calculated as follows:

$$weight(w) = \frac{1}{\sum_{i=1}^{5} N_i} \cdot (1.0 \times (N_1 + N_2) 0.75 \times N_3 + 0.5 \times (N_4 + N_5) \quad \text{(C.2)}$$

Where:

- N1: Number of appearance as the first word of a class name belonging to the provided interface.
- N2: Number of appearance in an entity name belonging to the provided interface of a component shape.
- N3: Number of appearance as the first word of an interface name.
- N4: Number of appearance other than the first word in an entity name.
- N5: Number of appearance in an entity name belonging to the center of a component shape.

C.3.3 Constructing the Component Name

In this step, a component name is constructed based on the strongest weighted tokens. The strongest weighted token constitutes the first word of the component name; the second strongest weighted word constitutes the second word of the component name and so on. The number of words used in a component name is chosen by the user. When many tokens have the same weight, all the possible combinations are presented to the user and he can choose the appropriate one.

List of Figures

List of Tables

Bibliography

[Acher *et al.*, 2011] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, et Philippe Lahire. Reverse engineering architectural feature models. In *5th European Conference on Software Architecture*, éditeurs Ivica Crnkovic, Volker Gruhn, et Matthias Book, volume 6903 de *LNCS*, pages 220–235, Essen, Germany, September 2011. Springer.

[Ajila et Kaba, 2004] Samuel Ajila et Badara Ali Kaba. Using traceability mechanisms to support software product line evolution. In *IRI*, pages 157–162. IEEE Systems, Man, and Cybernetics Society, 2004.

[Al-Msie'Deen *et al.*, 2013] Ra'Fat Al-Msie'Deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, et Hamzeh Eyal Salman. Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing. In *The 25th International Conference on Software Engineering and Knowledge Engineering*, page 8. Knowledge Systems Institute Graduate School, 2013.

[Ali *et al.*, 2011] Nasir Ali, Yann-Gaël andl Gueheneuc, et Giuliano Antoniol. Requirements traceability for object oriented systems by partitioning source code. In *18th Working Conf. on Reverse Engineering*, pages 45–54. IEEE Computer Society, 2011.

[Allier *et al.*, 2011] Simon Allier, Salah Sadou, Houari Sahraoui, et Regis Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, WICSA '11, pages 214–223, 2011.

[Anquetil *et al.*, 2010] Nicolas Anquetil, Uirá Kulesza, Ralf Mitschke, Ana Moreira, Jean-Claude Royer, Andreas Rummler, et André Sousa. A model-driven traceability framework for software product lines. *Softw. Syst. l.*, 9(4):427–451, 2010.

[Antoniol *et al.*, 2000] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, et Andrea De Lucia. Identifying the starting impact set of a maintenance request: A case study. In *CSMR*, pages 227–230, 2000.

[Antoniol *et al.*, 2002] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, et Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, Octobre 2002.

[Apel *et al.*, 2011] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, et Dirk Beyer. Detection of feature interactions using feature-aware verification. ASE '11, pages 372–375, Washington, DC, USA, 2011.

[Apel et Beyer, 2011] Sven Apel et Dirk Beyer. Feature cohesion in software product lines: an exploratory study. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 421–430, New York, NY, USA, 2011.

[Apiwattanapong *et al.*, 2005] Taweesup Apiwattanapong, Alessandro Orso, et Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 432–441, New York, NY, USA, 2005.

[Arnold, 1996] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[Asadi *et al.*, 2010] Fatemeh Asadi, Massimiliano Penta, Giuliano Antoniol, et Yann-Gael Gueheneuc. A heuristic-based approach to identify concepts in execution traces. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 31–40, Washington, DC, USA, 2010.

[Azmeh *et al.*, 2011] Zeina Azmeh, Fady Hamoui, Marianne Huchard, Nizar Messai, Chouki Tibermacine, Christelle Urtado, et Sylvain Vauttier. Backing composite web services using formal concept analysis. In *Proceedings of the 9th international conference on Formal concept analysis*, ICFCA'11, pages 26–41, Berlin, Heidelberg, 2011.

[Bachmann et Bass, 2001] Felix Bachmann et Len Bass. Managing variability in software architectures. SSR '01, pages 126–132, New York, NY, USA, 2001. ACM.

[Badri *et al.*, 2005] Linda Badri, Mourad Badri, et Daniel St-Yves. Supporting predictive change impact analysis: A control call graph based technique. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC '05, pages 167–175, Washington, DC, USA, 2005.

[Baeza-Yates et Ribeiro-Neto, 1999] Ricardo A. Baeza-Yates et Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Batory *et al.*, 2003] Don Batory, Jia Liu, et Jacob Neal Sarvela. Refinements and multi-dimensional separation of concerns. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 48–57, New York, NY, USA, 2003.

[Batory, 2005] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005.

[Benavides *et al.*, 2010] David Benavides, Sergio Segura, et Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[Bergroth *et al.*, 2000] Lasse Bergroth, Harri Hakonen, et Timo Raita. A survey of longest common subsequence algorithms. In *SPIRE*, pages 39–48, 2000.

[Berry *et al.*, 2014] Anne Berry, Alain Gutierrez, Marianne Huchard, Amedeo Napoli, et Alain Sigayret. Hermes: a simple and efficient algorithm for building the aoc-poset of a binary relation. *Annals of Mathematics and Artificial Intelligence*, pages 1–27, 2014.

[Beszédes *et al.*, 2007] Árpád Beszédes, Tamás Gergely, Szabolcs Farago, Tibor Gyimóthy, et Ferenc Fischer. The dynamic function coupling metric and its use in software evolution. In *CSMR*, pages 103–112. IEEE Computer Society, 2007.

[Bhatti *et al.*, 2012] Muhammad Usman Bhatti, Nicolas Anquetil, Marianne Huchard, et Stéphane Ducasse. A catalog of patterns for concept lattice interpretation in software reengineering. In *SEKE*, pages 118–123. Knowledge Systems Institute Graduate School, 2012.

[Bieman et Kang, 1995] James M. Bieman et Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the 1995 Symposium on Software Reusability*, SSR '95, pages 259–262, New York, NY, USA, 1995.

[Biggerstaff *et al.*, 1993] Ted J. Biggerstaff, Bharat G. Mitbander, et Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 482–498, Los Alamitos, CA, USA, 1993.

[bin Abid, 2009] Saad bin Abid. Resolving traceability issues in product derivation for software product lines. In *ICSOFT (1)*, pages 99–104. INSTICC Press, 2009.

[Binkley et Harman, 2005] David Binkley et Mark Harman. Locating dependence clusters and dependence pollution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance.*, ICSM'05, pages 177–186, Sept 2005.

[Bixin *et al.*, 2012] Li Bixin, Sun Xiaobing, Leung Hareton, et Zhang Sai. A survey of code-based change impact analysis techniques. In *software testing, verification and reliability*. Wiley Online Library, 2012.

[Black, 2001] Sue Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance*, 13(4):263–279, 2001.

[Blanc *et al.*, 2008] Xavier Blanc, Isabelle Mounier, Alix Mougenot, et Tom Mens. Detecting model inconsistency through operation-based model construction. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 511–520, New York, NY, USA, 2008.

[Bo Yu, 2004] Ali Mili Bo Yu. Requirements change impact in software architecture. WSEAS'04, pages 1–6, Miami, Florida, USA, 2004.

[Bogdan *et al.*, 2013] Dit Bogdan, Revelle Meghan, Gethers Malcom, et Poshyvanyk Denys. Feature location in source code: a taxonomy and survey. *Journal of Evolution and Process*, 25(1):53–95, 2013.

[Bohner, 2002] Shawn A. Bohner. Software change impacts - an evolving perspective. In *ICSM*, pages 263–272. IEEE Computer Society, 2002.

[Boldyreff *et al.*, 1996] Cornelia Boldyreff, Elizabeth Burd, R. M. Hather, Malcolm Munro, et E. J. Younger. Greater understanding through maintainer driven traceability. In *WPC*, pages 100–. IEEE Computer Society, 1996.

[Bosch, 2000] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[Bouktif *et al.*, 2006] Salah Bouktif, Yann-Gael Gueheneuc, et Giuliano Antoniol. Extracting change-patterns from cvs repositories. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 221–230, Washington, DC, USA, 2006.

[Breech *et al.*, 2006] Ben Breech, Mike Tegtmeyer, et Lori L. Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *ICSM*, pages 55–65. IEEE Computer Society, 2006.

[Briand *et al.*, 1999] Lionel C. Briand, Jürgen Wüst, et Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *ICSM*, pages 475–482, 1999.

[Briand *et al.*, 2009] L. C. Briand, Y. Labiche, et S. He. Automating regression test selection based on uml designs. *Inf. Softw. Technol.*, 51(1):16–30, 2009.

[Calder *et al.*, 2003] Muffy Calder, Mario Kolberg, Evan H. Magill, et Stephan Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Comput. Netw.*, 41(1):115–141, 2003.

[Chardigny *et al.*, 2008] Sylvain Chardigny, Abdelhak Seriai, Dalila Tamzalit, et Mourad Oussalah. Quality-driven extraction of a component-based architecture from an object-oriented system. In *CSMR*, pages 269–273. IEEE, 2008.

[Chechik *et al.*, 2009] Marsha Chechik, Winnie Lai, Shiva Nejati, Jordi Cabot, Zinovy Diskin, Steve Easterbrook, Mehrdad Sabetzadeh, et Rick Salay. Relationship-based change propagation: A case study. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, MISE '09, pages 7–12, Washington, DC, USA, 2009.

[Christian, 2007] Kästner Christian. Aspect-oriented refactoring of berkeley db. Master's thesis, University of Magdeburg, 2007.

[Churcher et Shepperd, 1995] Neville I. Churcher et Martin J. Shepperd. Towards a conceptual framework for object oriented software metrics. *SIGSOFT Softw. Eng. Notes*, 20(2):69–75, Avril 1995.

[Clements et Northrop, 2001] Paul C. Clements et Linda M. Northrop. *Software product lines: practices and patterns*. Addison-Wesley, 2001.

[Cornelissen *et al.*, 2009] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, et Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Software Eng.*, 35(5):684–702, 2009.

[Couto *et al.*, 2011] Marcus Vinicius Couto, Marco Tulio Valente, et Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 191–200, Washington, DC, USA, 2011.

[Czarnecki et Eisenecker, 2000] Krzysztof Czarnecki et Ulrich W. Eisenecker. Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[de Almeida, 2010] Rodrigo Bonif´acio de Almeida. *Modeling Software Product Line Variability in Use Case Scenarios*. PhD thesis, Universidade Federal de Pernambuco, 2010.

[De Lucia *et al.*, 2008] A. De Lucia, F. Fasano, et R. Oliveto. Traceability management for impact analysis. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 21–30, 2008.

[Deelstra *et al.*, 2004] Sybren Deelstra, Marco Sinnema, et Jan Bosch. Experiences in software product families: Problems and issues during product derivation. In *SPLC*, éditeur Robert L. Nord, volume 3154 de *Lecture Notes in Computer Science*, pages 165–182. Springer, 2004.

[Diana L. et Hassan, 2004] Webber Diana L. et Gomaa Hassan. Modeling variability in software product lines with the variation point model. *Sci. Comput. Program.*, 53(3):305–331, Décembre 2004.

[Dumais, 1992] Susan T. Dumais. Lsi meets trec: A status report. pages 137–152, 1992.

[Eaddy *et al.*, 2008] Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, et Yann-Gaél Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 53–62, Washington, DC, USA, 2008.

[Eisenbarth *et al.*, 2003] Thomas Eisenbarth, Rainer Koschke, et Daniel Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[Eisenberg et De Volder, 2005] Andrew David Eisenberg et Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 337–346, Washington, DC, USA, 2005.

[El Kharraz *et al.*, 2010] Amal El Kharraz, Petko Valtchev, et Hafedh Mili. Concept analysis as a framework for mining functional features from legacy code. In *Proceedings of the 8th International Conference on Formal Concept Analysis*, ICFCA'10, pages 267–282, Berlin, Heidelberg, 2010.

[Felix et Paul, 2005] Bachmann Felix et C. Clements Paul. Variability in software product lines. Technical report cmu/sei-2005-tr-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2005.

[Fluri *et al.*, 2005] Beat Fluri, Harald Gall, et Martin Pinzger. Fine-grained analysis of change couplings. In *SCAM*, pages 66–74. IEEE Computer Society, 2005.

[Fluri *et al.*, 2007] Beat Fluri, Michael Wuersch, Martin PInzger, et Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, 2007.

[Fluri et Gall, 2006] Beat Fluri et Harald C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 35–45, Washington, DC, USA, 2006.

[Furnas *et al.*, 1987] G. W. Furnas, T. K. Landauer, L. M. Gomez, et S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, 1987.

[Gall *et al.*, 2003] Harald Gall, Mehdi Jazayeri, et Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, IWPSE '03, pages 13–, Washington, DC, USA, 2003.

[Gallagher et Lyle, 1991] Keith B. Gallagher et James R. Lyle. Using program slicing in software maintenance. *IEEE Computer Society Trans. Software Engineering*, 17(8):751–761, August 1991.

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

[Gay *et al.*, 2009] Gregory Gay, Sonia Haiduc, Andrian Marcus, et Tim Menzies. On the use of relevance feedback in ir-based concept location. In *ICSM*, pages 351–360. IEEE, 2009.

[George et Andrea, 2004] Spanoudakis George et Zisman Andrea. Software traceability: A roadmap. In *Handbook of Software Engineering and Knowledge Engineering*, pages 395–428. World Scientific Publishing, 2004.

[Gîrba *et al.*, 2007] Tudor Gîrba, Stéphane Ducasse, Adrian Kuhn, Radu Marinescu, et Raţiu Daniel. Using concept analysis to detect co-change patterns. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 83–89, New York, NY, USA, 2007.

[Godin et Mili, 1998] Robert Godin et Hafedh Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, 1998.

[Gîrba *et al.*, 2005] Tudor Gîrba, Michele Lanza, et Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11. IEEE Computer Society, 2005.

[Gîrba et Ducasse, 2006] Tudor Gîrba et Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, 2006.

[Grossman et Frieder, 2004] David A. Grossman et Ophir Frieder. *Information Retrieval: Algorithms and Heuristics*. The Kluwer International Series of Information Retrieval. Springer, 2004.

[Gwizdala *et al.*, 2003] Steve Gwizdala, Yong Jiang, et Václav Rajlich. Jtracker - a tool for change propagation in java. In *CSMR*, pages 223–229, 2003.

[Haifeng et Zijie, 2010] Zhao Haifeng et Qi Zijie. Hierarchical agglomerative clustering with ordering constraints. In *WKDD*, pages 195–199, 2010.

[Halmans et Pohl, 2003] Günter Halmans et Klaus Pohl. Communicating the variability of a software-product family to customers. *Software and System Modeling*, 2(1):15–36, 2003.

[Hammad *et al.*, 2011] Maen Hammad, Michael L. Collard, et Jonathan I. Maletic. Automatically identifying changes that impact code-to-design traceability during evolution. *Software Quality Control*, 19(1):35–64, Mars 2011.

[Haslinger *et al.*, 2011] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, et Alexander Egyed. Reverse engineering feature models from programs' feature sets. WCRE '11, pages 308–312, Washington, DC, USA, 2011.

[Hattori *et al.*, 2008] Lile Hattori, Dalton Guerrero, Jorge Figueiredo, João Brunet, et Jemerson Damásio. On the precision and accuracy of impact analysis techniques. ICIS '08, pages 513–518, Washington, DC, USA, 2008.

[Hoffman, 2003] Michael A. Hoffman. Automated impact analysis of object-oriented software systems. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 72–73. ACM, 2003.

[Hutchins et Gallagher, 1998] Matthew A. Hutchins et Keith Gallagher. Improving visual impact analysis. In *ICSM*, pages 294–303, 1998.

[Ibrahim *et al.*, 2005] Suhaimi Ibrahim, Norbik Bashah Idris, Malcolm Munro, et Aziz Deraman. Integrating software traceability for change impact analysis. *Int. Arab J. Inf. Technol.*, 2(4):301–308, 2005.

[ISO, 2001] ISO. Software engineering – Product quality – Part 1: Quality model. Rapport Technique ISO/IEC 9126-1, International Organization for Standardization, 2001.

[Jain *et al.*, 1999] A. K. Jain, M. N. Murty, et P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[Jász *et al.*, 2008] Judit Jász, Árpád Beszedes, Tibor Gyimóthy, et Václav Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *ICSM*, pages 137–146. IEEE, 2008.

[Kagdi *et al.*, 2007] Huzefa H. Kagdi, Michael L. Collard, et Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, 19(2):77–131, 2007.

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

[Kang *et al.*, 1998] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, et Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, Janvier 1998.

[Kebir *et al.*, 2012a] Selim Kebir, Abdelhak-Djamel Seriai, Allaoua Chaoui, et Sylvain Chardigny. Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code. C3S2E '12, pages 1–8, New York, NY, USA, 2012.

[Kebir *et al.*, 2012b] Selim Kebir, Abdelhak-Djamel Seriai, Sylvain Chardigny, et Allaoua Chaoui. Quality-centric approach for software component identification from object-oriented code. WICSA-ECSA '12, pages 181–190, Washington, DC, USA, 2012.

[Khan. Simon Lock, 2009] Safoora Shakil Khan. Simon Lock. Concern tracing and change impact analysis: An exploratory study. In *Proceedings of the 2009 ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design*, EA '09, pages 44–48, Washington, DC, USA, 2009.

[Koschke et Quante, 2005] Rainer Koschke et Jochen Quante. On dynamic feature location. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 86–95, New York, NY, USA, 2005.

[Krueger, 2002] Charles W. Krueger. Easing the transition to software mass customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 282–293, London, UK, UK, 2002.

[Kuhn *et al.*, 2007] Adrian Kuhn, Stéphane Ducasse, et Tudor Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, Mars 2007.

[Kung *et al.*, 1994] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, et C. Chen. Change impact identification in object-oriented software maintenance. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 202–211, 1994.

[Kunrong et Václav, 2000] Chen Kunrong et Rajlich Václav. Case study of feature location using dependence graph. In *In Proceedings of the 8th International Workshop on Program Comprehension*, pages 241–249. IEEE Computer Society, 2000.

[Kuznetsov et Obiedkov, 2002] S. Kuznetsov et S. Obiedkov. Comparing performance of algorithms for generating concept lattices. *Journal of Experimental and Theoretical Artificial Intelligence*, 14:189–216, 2002.

[Law et Rothermel, 2003a] James Law et Gregg Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 430–, Washington, DC, USA, 2003.

[Law et Rothermel, 2003b] James Law et Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 308–318, Washington, DC, USA, 2003.

[Leung et White, 1991] H.K.N. Leung et L. White. A cost model to compare regression test strategies. In *Software Maintenance, Proceedings. Conference on ICSM*, pages 201–208, Oct 1991.

[Li et Offutt, 1996] L. Li et A.J. Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 171–184. IEEE Computer Society Press, 1996.

[Linden *et al.*, 2007] Frank J. van der Linden, Klaus Schmid, et Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Lindvall et Sandahl, 1996] Mikael Lindvall et Kristian Sandahl. Practical implications of traceability. *Softw. Pract. Exper.*, 26(10):1161–1180, 1996.

[Liu *et al.*, 2006] Jia Liu, Don Batory, et Christian Lengauer. Feature oriented refactoring of legacy applications. ICSE '06, pages 112–121, New York, NY, USA, 2006.

[Liu *et al.*, 2007] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, et Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 234–243, New York, NY, USA, 2007.

[Liu et Mei, 2003] Dongyun Liu et Hong Mei. Mapping requirements to software architecture by feature-orientation. In *STRAW*, pages 69–76, 2003.

[Lucia *et al.*, 2004] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, et Genoveffa Tortora. Enhancing an artefact management system with traceability recovery features. In *ICSM*, pages 306–315. IEEE Computer Society, 2004.

[Lucia *et al.*, 2007] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, et Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4), 2007.

[Lucia *et al.*, 2008a] Andrea De Lucia, R. Oliveto, et G. Tortora. Ir-based traceability recovery processes: An empirical comparison of "one-shot" and incremental processes. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 39–48, Washington, DC, USA, 2008.

[Lucia *et al.*, 2008b] Andrea De Lucia, Rocco Oliveto, et Genoveffa Tortora. Adams re-trace: traceability link recovery via latent semantic indexing. In *ICSE*, pages 839–842. ACM, 2008.

[Marcus *et al.*, 2004] Andrian Marcus, Andrey Sergeyev, Vàclav Rajlich, et Jonathan I. Maletic. An information retrieval approach to concept location in source code. *Reverse Engineering, Working Conference on*, 0:214–223, 2004.

[Marcus et Maletic, 2003a] Andrian Marcus et Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–137. IEEE Computer Society, 2003.

[Marcus et Maletic, 2003b] Andrian Marcus et Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–137. IEEE Computer Society, 2003.

[Marcus et Poshyvanyk, 2005] Andrian Marcus et Denys Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 133–142, Washington, DC, USA, 2005.

[Matinlassi, 2004] Mari Matinlassi. Comparison of software product line architecture design methods: Copa, fast, form, kobra and qada. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 127–136, Washington, DC, USA, 2004. IEEE Computer Society.

[Nakagawa *et al.*, 2011] Elisa Yumi Nakagawa, Pablo Oliveira Antonino, et Martin Becker. Reference architecture and product line architecture: A subtle but critical difference. In *ECSA*, éditeurs Ivica Crnkovic, Volker Gruhn, et Matthias Book, volume 6903 de *Lecture Notes in Computer Science*, pages 207–211. Springer, 2011.

[Orso *et al.*, 2003] Alessandro Orso, Taweesup Apiwattanapong, et Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. *SIGSOFT Softw. Eng. Notes*, 28(5):128–137, 2003.

[Oussalah, 2014] Mourad Oussalah. *Software Architecture*. Wiley-ISTE Ltd., France, Paris, 2014.

[Passos *et al.*, 2013] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wasowski, Christian stner, et Jianmei Guo. Feature-oriented software evolution. In *VaMoS '13*, pages 17:1–17:8, New York, NY, USA, 2013.

[Peng *et al.*, 2013] Xin Peng, Zhenchang Xing, Xi Tan, Yijun Yu, et Wenyun Zhao. Improving feature location using structural similarity and iterative graph mapping. *J. Syst. Softw.*, 86(3):664–676, Mars 2013.

[Petrenko *et al.*, 2008] M. Petrenko, V. Rajlich, et R. Vanciu. Partial domain comprehension in software evolution and maintenance. In *The 16th IEEE Int'l Conf. on Program Comprehension*, pages 13–22. IEEE, June 2008.

[Petrenko et Rajlich, 2009] Maksym Petrenko et Václav Rajlich. Variable granularity for improving precision of impact analysis. In *ICPC*, pages 10–19. IEEE Computer Society, 2009.

[Pirklbauer *et al.*, 2010] Guenter Pirklbauer, Christian Fasching, et Werner Kurschl. Improving change impact analysis with a tight integrated process and tool. In *ITNG*, pages 956–961. IEEE Computer Society, 2010.

[Pohl *et al.*, 2010] Klaus Pohl, Gnter Bckle, et Frank J. van der Linden. Software product line engineering: Foundations, principles and techniques. Springer Publishing Company, Incorporated, 2010.

[Poshyvanyk *et al.*, 2007] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, et Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Softw. Eng.*, 33(6):420–432, Juin 2007.

[Poshyvanyk *et al.*, 2009] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, et Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Softw. Engg.*, 14(1):5–32, 2009.

[Poshyvanyk et Marcus, 2007] Denys Poshyvanyk et Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. ICPC '07, pages 37–48, Washington, DC, USA, 2007.

[Rajlich, 1997] Václav Rajlich. A model for change propagation based on graph rewriting. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 84–91. IEEE Computer Society Press, 1997.

[Revelle *et al.*, 2011] Meghan Revelle, Malcom Gethers, et Denys Poshyvanyk. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical Softw. Engg.*, 16(6):773–811, Décembre 2011.

[Riebisch, 2003] Matthias Riebisch. Towards a more precise definition of feature models. In *Modelling Variability for Object-Oriented Product Lines*, éditeurs M. Riebisch, J. O. Coplien, et D. Streitferdt, pages 64–76. BookOnDemand Publ. Co, Norderstedt, 2003.

[Robbes *et al.*, 2007] Romain Robbes, Michele Lanza, et Mircea Lungu. An approach to software evolution based on semantic change. In *FASE*, volume 4422 de *Lecture Notes in Computer Science*, pages 27–41. Springer, 2007.

[Robbes07, 2007] Robbes07. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, January 2007.

[Robert et Matthias, 2008] Brcina Robert et Riebisch Matthias. Defining a traceability link semantics for design decision support. In *ECMDA-TW*, pages 39–48, 2008.

[Royer et Arboleda, 2012] Jean-Claude Royer et Hugh Arboleda. *Model-Driven and Software Product Line Engineering (ISTE)*. Wiley-IEEE Press, 1st édition, 2012.

[Rubin et Chechik, 2012] Julia Rubin et Marsha Chechik. Locating distinguishing features using diff sets. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 242–245, New York, NY, USA, 2012.

[Ryder et Tip, 2001] Barbara G. Ryder et Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 46–53, New York, NY, USA, 2001.

[Ryssel *et al.*, 2011] Uwe Ryssel, Joern Ploennigs, et Klaus Kabitzsch. Extraction of feature models from formal contexts. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 4:1–4:8, New York, NY, USA, 2011. ACM.

[Safyallah et Sartipi, 2006] Hossein Safyallah et Kamran Sartipi. Dynamic analysis of software systems using execution pattern mining. In *ICPC*, pages 84–88. IEEE Computer Society, 2006.

[Salton *et al.*, 1975] G. Salton, A. Wong, et C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[Salton et Buckley, 1988] Gerard Salton et Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.*, 24(5):513–523, aug 1988.

[Salton et McGill, 1986] Gerard Salton et Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[Savage *et al.*, 2010] Trevor Savage, Meghan Revelle, et Denys Poshyvanyk. Flat3: feature location and textual tracing tool. In *ICSE'2*, pages 255–258. ACM, 2010.

[Shao et Smith, 2009] Peng Shao et Randy K. Smith. Feature location by ir modules and call graph. In *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, pages 70:1–70:4, New York, NY, USA, 2009.

[Shen *et al.*, 2009] Liwei Shen, Xin Peng, et Wenyun Zhao. A comprehensive feature-oriented traceability model for software product line development. In *Australian Software Engineering Conference*, pages 210–219. IEEE Computer Society, 2009.

[Shepherd *et al.*, 2006] David Shepherd, Lori Pollock, et K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development*, AOSD '06, pages 3–14, New York, NY, USA, 2006.

[Shyam et Chris, 1994] Chidamber Shyam et Kemerer Chris. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, Jun 1994.

[Sochos *et al.*, 2006] Periklis Sochos, Matthias Riebisch, et Ilka Philippow. The feature-architecture mapping (farm) method for feature-oriented development of software product lines. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, ECBS '06, pages 308–318, Washington, DC, USA, 2006.

[Steffen, 2011] Lehnert Steffen. A review of software change impact analysis. Rapport technique, Technische Universität Ilmenau, 2011.

[Steven *et al.*, 2011] She Steven, Lotufo Rafael, Berger Thorsten, Wasowski Andrzej, et Czarnecki Krzysztof. Reverse engineering feature models. In *ICSE*, pages 461–470, 2011.

[Sun *et al.*, 2011] Xiaobing Sun, Bixin Li, Sai Zhang, Chuanqi Tao, Xiang Chen, et Wanzhi Wen. Using lattice of class and method dependence for change impact analysis of object oriented programs. In *SAC '11*, pages 1439–1444, New York, NY, USA, 2011.

[Sun et Li, 2011] Xiaobing Sun et Bixin Li. Using formal concept analysis to support change analysis. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 641–645, Washington, DC, USA, 2011.

[Susan *et al.*, 1990] T. Dumais Susan, W. Furnas George, K. Landauer Thomas, et Harshman Richard. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[Szyperski, 2002] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 2002.

[Thao *et al.*, 2008] Cheng Thao, Ethan V. Munson, et Tien N. Nguyen. Software configuration management for product derivation in software product families. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS '08, pages 265–274, Washington, DC, USA, 2008.

[Tonella, 2003] Paolo Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Trans. Softw. Eng.*, 29(6):495–509, 2003.

[Trinidad *et al.*, 2007] Pablo Trinidad, Antonio Ruiz Cortés, Joaquin Pena, et David Benavides. Mapping feature models onto component models to build dynamic software product lines. In *SPLC (2)*, pages 51–56. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007.

[Valtchev *et al.*, 2004] Petko Valtchev, Rokia Missaoui, et Robert Godin. Formal concept analysis for knowledge discovery and data mining: The new challenges. volume 2961 de *Lecture Notes in Computer Science*, pages 352–371. Springer, 2004.

[Vanciu et Rajlich, 2010] Radu Vanciu et Václav Rajlich. Hidden dependencies in software systems. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2010.

[Vidács *et al.*, 2007] László Vidács, Árpád Beszédes, et Rudolf Ferenc. Macro impact analysis using macro slicing. In *ICSOFT (SE)*, pages 230–235. INSTICC Press, 2007.

[Weiss et Lai, 1999] David M. Weiss et Chi Tau Robert Lai. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Wilde et Scully, 1995] Norman Wilde et Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[Xia et Srikanth, 2004] Franck Xia et Praveen Srikanth. A change impact dependency measure for predicting the maintainability of source code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*, COMPSAC '04, pages 22–23, Washington, DC, USA, 2004.

[Xiao *et al.*, 2007] Hua Xiao, Jin Guo, et Ying Zou. Supporting change impact analysis for service oriented business applications. In *Proceedings of the International Workshop on Systems Development in SOA Environments*, SDSOA '07, pages 6–, Washington, DC, USA, 2007.

[Xue *et al.*, 2012] Yinxing Xue, Zhenchang Xing, et Stan Jarzabek. Feature location in a collection of product variants. In *WCRE*, pages 145–154. IEEE Computer Society, 2012.

[Ying *et al.*, 2004] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, et Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.

[Yinxing *et al.*, 2010] Xue Yinxing, Xing Zhenchang, et Jarzabek Stan. Understanding feature evolution in a family of product variants. *Reverse Engineering, Working Conference on*, 0:109–118, 2010.

[Zalewski et Schupp, 2006] Marcin Zalewski et Sibylle Schupp. Change impact analysis for generic libraries. In *ICSM*, pages 35–44. IEEE Computer Society, 2006.

[Zave et Jackson, 1997] Pamela Zave et Michael Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, Janvier 1997.

[Zhang *et al.*, 2008] Jingjun Zhang, Xueyong Cai, et Guangyuan Liu. Mapping features to architectural components in aspect-oriented software product lines. CSSE '08, pages 94–97, Washington, DC, USA, 2008.

[Zhao *et al.*, 2006] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, et Fuqing Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, 2006.

[Ziadi *et al.*, 2012] Tewfik Ziadi, Luz Frias, Marcos Aurelio Almeida da Silva, et Mikal Ziane. Feature identification from the source code of product variants. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, éditeur FERENC R. MENS T., CLEVE A., pages 417–422, Los Alamitos, CA, USA, 2012.

[Zimmermann *et al.*, 2005] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, et Andreas Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.

[Zimmermann et Weißgerber, 2004] Thomas Zimmermann et Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. Int'l Workshop on Mining Software Repositories (MSR)*, pages 2–6, Los Alamitos CA, 2004.