THÈSE
Pour obtenir le grade de
Docteur

Délivré par l'**Université Montpellier II**

Préparée au sein de l'école doctorale **I2S***
Et de l'unité de recherche **UMR 5506**

Spécialité: **Informatique**

Présentée par **Ra'Fat AL-Msie'Deen**

Construction de lignes de produits logiciels par réingénierie de modèles de caractéristiques à partir de variantes de logiciels : l'approche REVPLINE

Soutenue le 10/07/2014 devant le jury composé de :

Mme. Marianne Huchard	Prof.	Université Montpellier II	Directeur de thése
M. Philippe Lahire	Prof.	Université de Nice Sophia Antipolis	Examinateur
M. Sébastien Ferré	Mdc.	Université de Rennes 1	Examinateur
M. Stefano Cerri	Prof.	Université Montpellier II	Président
M. Camille Salinesi	Prof.	Université de Paris 1	Rapporteur
M. Salah Sadou	Mdc.	Université de Bretagne Sud	Rapporteur
M. Abdelhak-Djamel Seriai	Mdc.	Université Montpellier II	Encadrant
Mme. Christelle Urtado	Mdc.	Ecole des Mines d'Alés	Encadrant
M. Sylvain Vauttier	Mdc.	Ecole des Mines d'Alés	Encadrant

* **I2S** : École doctorale Information Structures Systèmes

UNIVERSITÉ MONTPELLIER 2
SCIENCES ET TECHNIQUES

Laboratoire
Informatique
Robotique
Microélectronique
Montpellier

ACADÉMIE DE MONTPELLIER

UNIVERSITÉ MONTPELLIER II

— SCIENCES ET TECHNIQUES DU LANGUEDOC —

PH.D 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**

Reverse Engineering Feature Models From Software Variants to Build Software Product Lines

RIVEPLINE Approach

par

Ra'Fat AL-MSIE'DEEN

Soutenue le April 18, 2014, devant le jury composé de :

Mme. Marianne HUCHARD, Professeur, Université Montpellier II .... Directeur de Thèse
M. Philippe LAHIRE, Professeur, Université de Nice - Sophia Antipolis .... Examinateur
M. Sébastien FERRÉ, Maître de Conférences, Université de Rennes 1, ... Examinateur
M. Camille SALINESI, Université de Paris 1 .... Rapporteur
M. Salah SADOU, Maître de Conférences, Université de Bretagne Sud .... Rapporteur
M. Stefano CERRI, Professeur, Université Montpellier II .... Président
M. Abdelhak-Djamel SERIAI, Maître de Conférences, Université Montpellier II ....Co-Encadrant de Thése
M. Christelle URTADO, Maître de Conférences, Ecole des Mines d'Alés .... Co-Encadrant de Thése
M. Sylvain VAUTTIER, Maître de Conférences, Ecole des Mines d'Alés ....Co-Encadrant de Thése

Version of April 18, 2014

Contents

Contents iii

Acknowledgements vii

Abstract ix

Résumé xi

Personal Bibliography xiii

List of Abbreviations xv

1 Introduction: Context and Motivation 1
- 1.1 Context: Software Product Line . . . 2
- 1.2 Motivation and Problem . . . 2
- 1.3 Contribution . . . 5
- 1.4 Thesis Outline . . . 6

I Background and State of the Art 9

2 Background 11
- 2.1 Software Product Line and Software Variants . . . 12
  - 2.1.1 Software Product Line Engineering . . . 12
  - 2.1.2 Software Product Variants . . . 18
- 2.2 Formal and Relational Concept Analysis . . . 19
  - 2.2.1 Formal Concept Analysis . . . 20
  - 2.2.2 Relational Concept Analysis . . . 23
- 2.3 Latent Semantic Indexing . . . 26
  - 2.3.1 The Singular Value Decomposition Model . . . 28
  - 2.3.2 Singular Value Decomposition Numerical Example . . . 28
  - 2.3.3 Evaluation Metrics . . . 31
  - 2.3.4 Latent Semantic Indexing Through Example . . . 32
- 2.4 Conclusion . . . 35

3 State of the Art 37
- 3.1 Introduction . . . 38
- 3.2 Key Concepts . . . 38
  - 3.2.1 Feature Location . . . 38
  - 3.2.2 Source Code Documentation . . . 39
  - 3.2.3 Reverse Engineering Feature Model . . . 39
- 3.3 Related work . . . 40

3.3.1 Feature Location Approaches . . . 40
3.3.2 Source Code Documentation Approaches . . . 50
3.3.3 Reverse Engineering FMs Approaches . . . 59
3.4 Summary . . . 68

II RIVEPLINE Approach: Contributions 71

4 REVPLINE: Feature Location in a Collection of Software Product Variants 73
4.1 Presentation of the Problem . . . 74
4.2 Principles of the Proposal . . . 75
4.2.1 Goal and Core Assumptions . . . 75
4.2.2 Object-oriented Source Code Variation . . . 76
4.2.3 Features versus Object-oriented Building Elements: the Mapping Model . . . 77
4.2.4 The Lexical Versus Structural Similarity . . . 79
4.2.5 An Illustrative Example: Drawing Shapes Software Variants . . . 80
4.3 The Feature Mining Process into Details . . . 81
4.3.1 Identifying the Common Block and Blocks of Variation . . . 82
4.3.2 Identifying Atomic Blocks . . . 83
4.4 Threats to validity . . . 92
4.5 Conclusion . . . 94

5 REVPLINE: Documenting the Mined Feature Implementation 97
5.1 Introduction . . . 98
5.2 Specify Use-case Diagrams of Software Variants with Variability . . . 99
5.2.1 Exploiting Use-cases to Support Feature Documentation . . . 99
5.2.2 An Illustrative Example: Mobile Tourist Guide Software Variants . . . 101
5.3 Principles of Feature Documentation . . . 102
5.4 Feature Documentation Overview . . . 104
5.5 Feature Documentation Step by Step . . . 106
5.5.1 Reduce LSI Search Space: Identifying Hybrid Blocks Based on RCA . . . 106
5.5.2 Exploring the Hybrid Blocks CLF to Identify Features Documentation . . . 108
5.5.3 Measuring Hybrid block contents' Similarity Based on LSI . . . 109
5.5.4 Identifying Feature Name and Description Based on FCA . . . 112
5.6 Naming Feature Implementation Based on OBE Names . . . 114
5.7 Threats to validity . . . 115
5.8 Conclusion . . . 116

6 Reverse Engineering Feature Models from Software Configurations 117
6.1 Introduction . . . 118
6.2 FM Reverse Engineering Process . . . 119
6.3 Step-by-Step FM Reverse Engineering . . . 121
6.3.1 Extracting the AOC-poset . . . 121
6.3.2 Extracting root feature . . . 123
6.3.3 Extracting mandatory features . . . 123
6.3.4 Extracting atomic set of features (AND-group) . . . 124
6.3.5 Extracting exclusive-or relation . . . 124
6.3.6 Extracting inclusive-or relation . . . 126
6.3.7 Extracting requires constraint . . . 127
6.3.8 Extracting Exclude constraint . . . 127
6.3.9 The Resulting Feature Model . . . 128
6.4 FM Evaluation . . . 129

6.5 Conclusion . . . 131

III Experimentation **133**

7 Experimentation **135**
7.1 Introduction . . . 136
7.2 ArgoUML-SPL Case Study . . . 136
7.2.1 ArgoUML-SPL Description . . . 136
7.2.2 ArgoUML Validation . . . 138
7.3 Health complaint-SPL Case Study . . . 145
7.3.1 Health complaint-SPL Description . . . 145
7.3.2 Health Complaint Validation . . . 148
7.4 Mobile Media Case Study . . . 155
7.4.1 Mobile Media Description . . . 155
7.4.2 Mobile Media Validation . . . 157
7.5 Reverse Engineering FMs from Samples of Program Configurations . . . 163
7.6 Conclusion . . . 164

IV Conclusion and Perspectives **167**

8 Conclusion and Future Directions **169**
8.1 Summary of Contributions . . . 170
8.2 Future Directions . . . 172
8.2.1 Extending the Scope . . . 172
8.2.2 Improving the Approach with Natural Language Processing Tools . . . 173
8.2.3 Improving techniques . . . 173

A Implementation **175**
A.1 Introduction . . . 176
A.2 REVPLINE Components . . . 176
A.3 External Libraries . . . 182
A.3.1 eRCA . . . 182
A.3.2 Graphviz . . . 184
A.4 Eclipse Plugins . . . 185
A.4.1 UML2 Plugin . . . 185
A.4.2 FeatureIDE Plugin . . . 186
A.5 Statistical information regarding REVPLINE source code . . . 187
A.6 Conclusion . . . 189

List of Figures **190**

List of Tables **192**

List of Listings **194**

Bibliography **195**

Acknowledgements

When you make the finding yourself - even if you're the last person on Earth to see the light - you never forget it.
Carl Sagan

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. First, I would like to thank my supervisors: *Marianne Huchard*, *Christelle Urtado*, *Abdelhak-Djamel Seriai* and *Sylvain Vauttier*. They brought me into the domain of software engineering and software product line engineering. They taught me in some many aspects: paper writing, presentation skills, or even programming skills. And thanks to their guidance and encouragement, I found the suitable topic and learned the methods to do research. I would also like to thank the thesis committee members, for their time in reading and commenting on my thesis. To the MaREL group, I say: Thank you!

Thanks to my families' support, that enable me to focus on my research. Especially, I would thank my mother and father, who encouraged me a lot when my research progress did go well. And my brothers also supported and encouraged me a lot. They also helped to take care of my parents when I was busy with my research work. For my family, I also wish my PhD thesis is a gift to them.

I am deeply and forever indebted to the people in my life that touched my heart and gave me strength to move forward to something better. The people who inspire me to breathe, who encourage me to understand who I am, and who believe in me when no one else does. Am also thankful to all my colleagues and friends in Jordan and France, especially from the Tafila Technical University and University Montpellier 2 for their help and support, with whom I shared pleasant times.

LIRMM — 161, rue Ada — 34095 Montpellier cedex 5 — France

Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM)

Équipe MaREL — LIRMM @ UM2

Abstract

framework tackles the consumer related part of Web service utilization, including Web service discovery, selection for independent or composition utilization, and substitution. Therefore, each layer of our framework corresponds to a level of the described problem in the previous section, and they are as follows:

The idea of Software Product Line (SPL) approach is to manage a family of similar software products in a reuse-based way. Reuse avoids repetitions, which helps reduce development/maintenance effort, shorten time-to-market and improve overall quality of software. To migrate from existing software product variants into SPL, one has to understand how they are similar and how they differ one from another. Companies often develop a set of software variants that share some features and differ in other ones to meet specific requirements. To exploit existing software variants and build a software product line, a feature model must be built as a first step. To do so, it is necessary to extract mandatory and optional features in addition to associate each feature with its name. Then, it is important to organize the mined and documented features into a feature model. In this context, our thesis proposes three contributions.

Thus, we propose, in this dissertation as a first contribution a new approach to mine features from the object-oriented source code of a set of software variants based on Formal Concept Analysis, code dependency and Latent Semantic Indexing. The novelty of our approach is that it exploits commonality and variability across software variants, at source code level, to run Information Retrieval methods in an efficient way. The second contribution consists in documenting the mined feature implementations based on Formal Concept Analysis, Latent Semantic Indexing and Relational Concept Analysis. We propose a complementary approach, which aims to document the mined feature implementations by giving names and descriptions, based on the feature implementations and use-case diagrams of software variants. The novelty of our approach is that it exploits commonality and variability across software variants, at feature implementations and use-cases levels, to run Information Retrieval methods in an efficient way. In the third contribution, we propose an automatic approach to organize the mined documented features into a feature model. Features are organized in a tree which highlights mandatory features, optional features and feature groups (and, or, xor groups). The feature model is completed with requirement and mutual exclusion constraints. We rely on Formal Concept Analysis and software configurations to mine a unique and consistent feature model. To validate our approach, we applied it on three case studies: ArgoUML-SPL, Health complaint-SPL, Mobile media software product variants. The results of this evaluation validate the relevance and the performance of our proposal as most of the features and its constraints were correctly identified.

Keywords: Software Product Line Engineering, Software Product Variants, Re-engineering, Feature location, Feature model, Variability, Formal Concept Analysis, Latent Semantic Indexing, Relational Concept Analysis, Feature documentation, Code comprehension, Use-case diagram.

Résumé

Les lignes de produits logicielles constituent une approche permettant de construire et de maintenir une famille de produits logiciels similaires mettant en œuvre des principes de réutilisation. Ces principes favorisent la réduction de l'effort de développement et de maintenance, raccourcissent le temps de mise sur le marché et améliorent la qualité globale du logiciel. La migration de produits logiciels similaires vers une ligne de produits demande de comprendre leurs similitudes et leurs différences qui s'expriment sous forme de caractéristiques (features) offertes. Dans cette thèse, nous nous intéressons au problème de la construction d'une ligne de produits à partir du code source de ses produits et de certains artefacts complémentaires comme les diagrammes de cas d'utilisation, quand ils existent. Nous proposons des contributions sur l'une des étapes principales dans cette construction, qui consiste à extraire et à organiser un modèle de caractéristiques (feature model) dans un mode automatisé.

La première contribution consiste à extraire des caractéristiques dans le code source de variantes de logiciels écrits dans le paradigme objet. Trois techniques sont mises en œuvre pour parvenir à cet objectif : l'Analyse Formelle de Concepts, l'Indexation Sémantique Latente et l'analyse des dépendances structurelles dans le code. Elles exploitent les parties communes et variables au niveau du code source. La seconde contribution s'attache à documenter une caractéristique extraite par un nom et une description. Elle exploite le code source mais également les diagrammes de cas d'utilisation, qui contiennent, en plus de l'organisation logique des fonctionnalités externes, des descriptions textuelles de ces mêmes fonctionnalités. En plus des techniques précédentes, elle s'appuie sur l'Analyse Relationnelle de Concepts afin de former des groupes d'entités d'après leurs relations. Dans la troisième contribution, nous proposons une approche visant à organiser les caractéristiques, une fois documentées, dans un modèle de caractéristiques. Ce modèle de caractéristiques est un arbre étiqueté par des opérations et muni d'expressions logiques qui met en valeur les caractéristiques obligatoires, les caractéristiques optionnelles, des groupes de caractéristiques (groupes ET, OU, OU exclusif), et des contraintes complémentaires textuelles sous forme d'implication ou d'exclusion mutuelle. Ce modèle est obtenu par analyse d'une structure obtenue par Analyse Formelle de Concepts appliquée à la description des variantes par les caractéristiques. L'approche est validée sur trois cas d'étude principaux : ArgoUML-SPL, Health complaint-SPL et Mobile media.

Mots clefs: Ingénierie des lignes de produits, variante de logiciel, Réingénierie, identification de caractéristique, modèle de caractéristiques, Variabilité, Analyse Formelle de Concepts, Indexation Sémantique Latente, Analyse Relationnelle de Concepts, Documentation de caractéristiques, Compréhension du code, Diagramme de cas d'utilisation.

Personal Bibliography

Search for the truth is the noblest occupation of man, its publication is a duty.
Anne Louise Germaine de Staël-Holstein

This research activity has led to several publications. This thesis reuses and extends publications of the author. These publications are summarized below.

International Conferences:

❶ R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Feature location in a collection of software product variants using formal concept analysis,". In Proceedings of the 13th International Conference on Software Reuse (ICSR), ser. Lecture Notes in Computer Science, J. M. Favaro and M. Morisio, Eds., vol. 7925. Springer, 2013, pp. 302–307.

❷ R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing,". In Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering (SEKE). Knowledge Systems Institute Graduate School, 2013, pp. 244–249.

❸ R. Al-Msie'deen, A.-D. Seriai, M. Huchard, C. Urtado, and S. Vauttier, "Mining features from the object-oriented source code of software variants by combining lexical and structural similarity,". In Proceedings of the 14th International Conference on Information Reuse and Integration (IRI). IEEE, 2013, pp. 586–593.

❹ R. Al-Msie'deen, M. Huchard, A.-D. Seriai, C. Urtado, S. Vauttier, and A. Al-Khlifat, "Concept lattices: a representation space to structure software variability,". In Proceedings of the 5th International Conference on Information and Communication Systems (ICICS). IEEE, 2014, pp. 72–77.

Book:

➟ R. Al-Msie'deen, A. Seriai, and M. Huchard, Reengineering Software Product Variants Into Software Product Line: REVPLINE Approach. LAP Lambert Academic Publishing, 2014, pp. 1-120.

PHD Symposium:

➟ R. Al-Msie'deen, "Mining feature models from the object-oriented source code of a collection of software product variants,". In Doctoral Symposium - The 27th European Conference on Object-Oriented Programming (ECOOP), 2013, pp. 1–10.

National Conference:

➟ R. AL-Msie'deen, A. D. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "An approach to recover feature models from object-oriented source code,". In Actes de la Journée Lignes de Produits 2012, Lille, France, Novembre 2012, pp. 15–26.

Poster:

➠ R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "A methodology to recover feature models from object-oriented source code," 2012, poster presented at VARY Workshop (VARY: VARiability for You @ MODELS 2012), September 30, Innsbruck, Austria.

International Workshops:

❶ H. Eyal-Salman, A.-D. Seriai, C. Dony, and R. Al-msie'deen, "Recovering traceability links between feature models and source code of product variants,". In Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone, ser. VARY '12. New York, NY, USA: ACM, 2012, pp. 21–25.

❷ H. Eyal-Salman, A.-D. Seriai, C. Dony, and R. Al-msie'deen, "Identifying traceability links between product variants and their features,". In Proceedings of the First International Workshop on Reverse Variability Engineering, ser. REVE '13, 2013, pp. 17–23.

List of Abbreviations

REVPLINE	REengineering Software Product Variants into Software Product LINE
SPLE	Software Product Line Engineering
SPL	Software Product Line
FMs	Feature Models
FCA	Formal Concept Analysis
RCA	Relational Concept Analysis
RCF	Relational Context Family
CLF	Concept Lattice Family
IR	Information Retrieval
LSI	Latent Semantic Indexing
VSM	Vector Space Model
LSA	Latent Semantic Analysis
FODA	Feature Oriented Domain Analysis
LoC	Lines of Code
CB	Common Block
BV	Block of Variation
CAB	Common Atomic Block
ABV	Atomic Blocks of Variation
LSM	Lexical Similarity Matrix
CM	Combined Matrix
DSM	Dependency Structure Matrix

CHAPTER 1

Introduction: Context and Motivation

The past is but the beginning of a beginning, and all that is or has been is but the twilight of the dawn.
Herbert G. Wells

Preamble

This chapter introduces the context, motivation, problem statement and contribution of our proposal. Section 1.1 presents the context of our work. The context is software engineering and more specifically software product line engineering. In Section 1.2, we present the motivation of our proposal. This section also explains the problems regarding the existing approaches in this domain and the strategies they are using to build a software product line and this is what we do in this dissertation. In Section 1.3, we introduce our contribution regarding the problem statements. Finally, in Section 1.4, we present characteristics of the contributions and organization of the document.

1.1 Context: Software Product Line

Similarly to car developers who propose a full range of cars with common characteristics and numerous variants, software developers may cater to various needs and propose as a result a software family instead of one single product. Such software family is called a software product line (SPL) [Clements and Northrop, 2002]. The SPL approach targets at improving software productivity and quality by relying on the similarity that exists among software systems and related development process. The idea of the SPL approach is to manage a family of software systems in a reuse-based way. The motivation of SPL lies in the fact that companies most of the time develop and maintain multiple variants of the same software system customized for the needs of different customers. All such system variants are namely similar, but they also differ in customer-specific features [Xue *et al.*, 2012]. This forms possibility for reuse. Reuse avoids duplications, which helps decrease development and maintenance effort and cost, shorten time-to-market and improves quality of software [Jacobson *et al.*, 1997] [Xue, 2011].

In an SPL, core assets [Bass *et al.*, 2003] [Clements and Northrop, 2002] are identified and built. Product variants are derived from core assets. Variability among software variants is described in terms of features [Kang, 1990]. Constraints between features are described via Feature Models (FMs). FMs are the *de facto* standard to model the mandatory and optional features of an SPL and their relationships [Acher *et al.*, 2013b].

A SPL is usually characterized by two sets of features: the features that are shared by all products in the family, which represent the *SPL's commonality*, and the features that are shared by some, but not all, products in the family, which represent the *SPL's variability*.

1.2 Motivation and Problem

In common software development processes software product variants often evolve from an initial product developed for and successfully used by the first customer. Mobile Media Systems [Tizzei *et al.*, 2011] are an example of such a product evolution. These product variants usually share some common features but they are also different from one another due to subsequent customization to meet the specific requirements of different customers [Xue *et al.*, 2012].

As the number of features and the number of software product variants grows, it is worth reengineering software product variants into a SPL for systematic reuse [Clements and Northrop, 2002]. SPLs are often set up after the implementation of numerous similar software product variants using *ad hoc* reuse techniques such as copy-paste-modify [Ziadi *et al.*, 2012]. In practice, most organizations cannot afford to start a SPL development from scratch and therefore have to use as much existing software assets (source code, design artefacts) as possible [Beuche, 2009]. When variants become numerous, switching to a rigorous software product line engineering (SPLE) process is a solution to tame the increasing complexity of all the engineering tasks. To switch to SPLE starting from a collection of existing variants, the first step is to mine the FM that describes the SPL. This implies to identify the software family's common and variable features. Manual reverse engineering of a feature model for software variants is time-consuming, error-prone, and requires substantial efforts [Ziadi *et al.*, 2012].

Building a SPL from existing software product variants, a number of open problems must be solved. Those open problems include how to discover or extract the variability amongst the software product variants, how to model variability and commonality, how to build the real FM that expresses the variability and commonality across these software variants.

In order to re-engineer existing software variants into a SPL, numerous important fundamentals must be satisfied [van der Linden *et al.*, 2007]. Variability and commonality amongst the software variants must be clearly identified and should be systematically managed via FM [Xue, 2011]. In the following we present the sub-problems that we identified as important in order to answer the question: "How do we want to re-engineer the code of software variants into a SPL?".

❶ Mining features from the source code of software variants: In order to re-engineer software variants into a SPL, there is a need to extract all features from these variants' source code. Source code of software variants is considered as the most important source of information. To mine software variants feature we must distinguish between two types of features, optional (*i.e.* variable) and mandatory (*i.e.* common) features. Mandatory features appear in all software variants, while optional features appear in some but not all software variants.

Many approaches presented for feature location in a single software system. In our work, we focus on feature location in a collection of software product variants. The existing approaches fail to address efficiently this problem (*i.e.* feature location in a collection of software variants). The majority of existing approaches are designed to locate the program elements of a particular feature in a single software system. In the context of software product variants, most of the existing approaches investigate variability at the package or class levels. In addition, these approaches accept as input two sources of information such as source code of software variants and feature/feature description.

The majority of existing works (*cf.* Chapter 3) identifies the traceability link between features and source code of a single software system [Rubin and Chechik, 2013b]. An inclusive survey about approaches linking features and source code in single software is proposed in [Dit *et al.*, 2013]. The identification of relationships (*i.e.* traceability links) between use-case diagrams and source code of a single software is the subject of the work by Grechanik *et al.* [Grechanik *et al.*, 2007]. In [Marcus and Maletic, 2003], the authors propose an approach to recover documentation-to-source-code traceability links using Latent Semantic Indexing (LSI) in a single software system.

Existing work linking features and source code in collections of software variants such as [Linsbauer *et al.*, 2013] [Xue *et al.*, 2012] [Salman *et al.*, 2013] rely on pre knowledge, where these approaches use software variants features and features descriptions as input in addition to the source code itself. In [Xue, 2011], the authors provide an approach that can automatically and systematically recover the variability at both the requirement and the code implementation levels. The proposed approach integrates model differencing, clone detection and information retrieval techniques. The proposed approach uses as input in addition to the source code the feature set and feature descriptions to recover Feature-to-Code traceability links in a software product family [Xue *et al.*, 2012]. In our work, we only rely on the source code as input of the mining process (*i.e.* we don't know features in advance).

Existing approaches that extract features from software variants source code only such as [Ziadi *et al.*, 2012] and [Rubin and Chechik, 2013a] have some limits. In [Ziadi *et al.*, 2012], the authors propose an approach to extract commonality and variability at source code level from existing SPL. In their work, they never distinguish amongst mandatory features (*i.e.* their approach gathers all mandatory features as a single mandatory feature under a title base feature). Their approach does not distinguish between the feature implementation for those set of features (*i.e.* optional features) that always appear together in a set of product variants. On the other hand, their work only considers software variants in which the variability represents at package, class, attribute and method level. Their work does not consider software variants where variability is mainly present at method body level. In [Rubin and Chechik, 2012], authors present an approach to locate optional features from two software variants' source code as one set of source code (*i.e.* diff set). They do not consider common features. They also are limited to only two software variants. They do not distinguish between the several optional features (*i.e.* their approach gather all optional features as a single feature).

❷ Documenting the mined feature implementations: The implementation of each feature in the software variants source code may correspond to a huge number of source code elements; the mined feature must be documented. The goal of this documentation is to reflect feature roles at the domain level. Additionally, for purposes of constructing a FM and reusing existing features in other software, each feature implementation that is presented to the human user must have a meaningful name. In addition, feature documentation is needed in order to understand existing software variants and facilitate their maintenance. Feature documentation means giving a name and a description for the mined feature implementation.

The majority of existing approaches (*cf.* Chapter 3) are designed to extract labels, names, topics or code summarization in a single software system. The majority of existing approaches manually assign feature names to the feature implementations. In the literature there is no work which gives a name or description for the mined feature implementation.

Existing approaches extract labels, names or topics from the source code of single software and they aim to facilitate source code comprehension. Kuhn *et al.* [Kuhn, 2009] present a lexical approach to automatically provide labels for components of a single software. Kebir *et al.* [Kebir *et al.*, 2012] propose an approach to identify components from the object-oriented source code of a single software and provide names for the identified components. De Lucia *et al.* [Lucia *et al.*, 2012] propose an approach to extract labels from the source code of a single software, based on Information Retrieval (IR) techniques. Falleri *et al.* [Falleri *et al.*, 2010] present a wordNet-like approach to extract the structure of a single software by using the relationships among identifier names (*e.g.* packages, classes, methods, variables, *etc.*). Haiduc *et al.* [Haiduc *et al.*, 2010] propose a technique for automatically summarizing source code of single software. Kuhn *et al.* [Kuhn *et al.*, 2007] use a Latent Semantic Indexing (LSI) based approach for identifying topics in source code of a single software by semantically clustering software artifacts such as methods, files or packages based on identifier names and comments.

In case of software variants, no one proposes name or description for the mined feature implementations. Ziadi *et al.* [Ziadi *et al.*, 2012] propose an approach to identify features across software variants. In their work, they propose manually the feature names. In [Yang *et al.*, 2009], the authors analyze open source applications with similar functionality in order to extract an initial FM. In their work, they manually propose the feature name for each cluster (by analysts). Davril *et al.* [Davril *et al.*, 2013] present an approach to construct FMs from product descriptions. They develop a cluster-naming process that involves selecting the most frequently occurring phrase among all of the feature descriptors in the cluster. This work deals with product descriptions, not with source code. In [Xue *et al.*, 2012] [Linsbauer *et al.*, 2013] [Salman *et al.*, 2013], the authors recover Feature-to-Code traceability links in a software product family. In their work, the feature name and its description was known in advance. In their work there is no assumption about how features are named, described and expressed.

❸ Expressing the mined and documented feature as feature model: The FM represents valid software configurations in addition to constraints between features (*i.e.* a feature A requires or excludes feature B). The mined FM and core assets (*i.e.* feature implementations) from software variants represent the core of the SPL.

The majority of existing approaches (*cf.* Chapter 3) are designed to reverse engineering FM from high level models (*e.g.* product description and requirements) and other approaches deal directly with low level model (*i.e.* source code) with a lot of limitations. Some approaches offer a solution acceptable but not able to identify important parts of feature model such as cross-tree constraints (require and exclude), and-group, or-group and xor-group.

Many approaches propose to extract FMs from different artifacts such as components of single software, variability descriptions, product descriptions, product configurations, feature sets, domain application's source code, textual feature descriptions and feature dependencies, requirements or incidence matrix.

In [Paškevičius *et al.*, 2012], authors present a framework for the automated derivation of features and FMs from the existing software artefacts (components, libraries, classes, *etc.*). They extract a basic

FM without dependencies between its features. The extracted FM is from a single software. In [Acher *et al.*, 2013b] [Acher *et al.*, 2011], the authors propose a reverse engineering process for producing a variability model (*i.e.* a feature model) of a plugin-based architecture. They develop automated techniques to extract and combine different variability descriptions, including a hierarchical software architecture model, a plugin dependency model and the software architect's knowledge. In [Acher *et al.*, 2012], authors present an approach to synthesize a FM based on the product descriptions. Their approach takes as its input product descriptions for a collection of product variants to build the FM. Products are described by characteristics (language, license, *etc.*) with different patterns on values (many-valued, one-valued, etc.).

In [Davril *et al.*, 2013], authors present a novel algorithm for automating the generation of a FM from a set of informal and incomplete product descriptions. In [Loesch and Ploedereder, 2007], the authors present a method for restructuring and simplifying the provided variability in an SPL. Their work does not produce FMs. They apply Formal Concept Analysis (FCA) to analyze the variability in a SPL based on product configurations (described by features), and construct a concept lattice that provides a classification of the used features. They identify the exclusive-or relation, in addition to cross-tree constraints (*i.e.* includes and requires constraints). In [Ryssel *et al.*, 2011], authors extract a basic FM without constraints (dependencies) from an incidence matrix describing products by their characteristics. They identify the exclusive-or/inclusive-or relations.

In [Yang *et al.*, 2009], authors analyze open source applications for multiple existing domain applications with similar functionalities. Therefore, they use the data access semantics of program units (limited only to methods) as the analysis basis, supported by the data schema mapping among different applications. They recover initial domain FMs using data access semantics, FCA, concept pruning/merging, structure reconstruction and variability analysis. Their approach is limited only to three applications. In [She *et al.*, 2011], authors propose a reverse engineering approach combining two distinct sources of information: textual feature descriptions and feature dependencies. They developed an efficient synthesis procedure to compute variability information (*i.e.* feature groups) and proposed heuristics for identifying the most likely parent feature candidates of each feature.

In [Weston *et al.*, 2009] authors develop a tool which creates FMs from natural language requirements specifications. First, they divided the specifications in fragments and then used clustering techniques to identify features. In [Chen *et al.*, 2005] authors propose an approach to build FMs from application specifications. Authors introduce a classification of relationships between requirements. For each application, the procedure first elicits a set of functional requirements and models the relationships between them in an undirected graph. Features are then identified by clustering the functional requirements. Finally, the resulting FMs a merged as one. Some approaches extract FM from product configurations (*i.e.* feature sets) such as [Haslinger *et al.*, 2011] [Acher *et al.*, 2013a] [Acher *et al.*, 2012]. The main challenge of these works is that numerous candidate FMs can be extracted from the same input configurations, yet only a few of them are meaningful and maintainable.

Companies often develop a set of software variants that share some features and differ in other ones to meet specific requirements. To exploit existing software variants and build a software product line, a feature model must be built as a first step. To do so, it is necessary to extract mandatory and optional features in addition to associate each feature with its name. Then, It is important to organize the mined and documented features into a feature model: features are organized in a tree which highlights mandatory features, optional features and feature groups (and, or, xor groups). The feature model is completed with require constraints and mutual exclusion constraints.

1.3 Contribution

To exploit existing variants and build a SPL, a feature model of this SPL must be built as a first step. To do so, it is necessary to mine mandatory and optional features from the source code of software variants, in addition to document each feature implementation by assigning its name. The proposed approach takes as its input the source code and use-case diagrams of software variants. The first step is to extract the feature implementations through these variants. The second step is to document the mined feature implementations using use-case diagrams (or identifier names). The third step is to extract the FM with its constraints from software configurations (*i.e.* from the mined and documented features). In this section, we summarize our main contributions.

❶ Contrarily to previous work which mainly address feature location in a *single software system*, we focus on locating features from object-oriented source code of a collection of software systems. To locate optional and mandatory features across the source code of software variants we propose the REVPLINE[1] approach. The novelty of our approach is that we exploit commonality and variability across the source code of software variants, to apply Information Retrieval (IR) methods in an efficient way. In the REVPLINE approach, we rely on lexical (*i.e.* textual) and structural similarity to mine feature implementation as an atomic set of source code elements. The REVPLINE approach considers software variants in which the variability is represented at different levels of source code elements such as the name of packages, classes, attributes, methods, and method body elements such as local variables, method invocations and attribute accesses. The REVPLINE feature location approach uses two techniques : Formal Concept Analysis (FCA) and Latent Semantic Indexing (LSI) (*cf.* Chapter 4).

❷ We extend the REVPLINE approach to document the mined feature implementations. Previous work which extracts labels or topics deal with a single software. In the context of software variants, previous work manually named the extracted feature implementations. In our work, the documentation of the mined feature implementation is done in two ways. Firstly, we document the mined features based on the names of the source code elements themselves (*i.e.* identifier names). Secondly, we document the mined features based on the feature implementations and use-case diagrams of software variants. Our approach gives each feature implementation a name and description based on the use-case name and description. The novelty of our approach is that we exploit commonality and variability across software variants, at feature implementation and use-cases levels, to apply Information Retrieval (IR) methods in an efficient way. The feature documentation process takes the variants' use-cases and the mined feature implementations as its inputs. REVPLINE documentation approach uses three techniques : Formal Concept Analysis (FCA), Latent Semantic Indexing (LSI) and Relational Concept Analysis (RCA) (*cf.* Chapter 5).

❸ In this dissertation, we propose an automatic approach to organize the mined documented features into a feature model. Features are organized in a tree which highlights mandatory features, optional features and feature groups (and, or, xor groups). The feature model is completed with requirement and mutual exclusion constraints. We rely on Formal Concept Analysis and software configurations to mine a unique and consistent feature model. Our technique constructs a concept lattice from product configurations. The constructed lattice provides a classification of the mandatory and optional features in addition to the constraints between these features. Our technique could be used to extract a FM that expresses the same set of valid product configurations in a way that may be easier to understand for the human reader or manager (*cf.* Chapter 6).

1.4 Thesis Outline

The remainder of this dissertation is structured to gradually present precise definitions for the concepts informally addressed in this introductory chapter in addition to the REVPLINE approach. The contents of the remaining chapters are as follows:

[1]REVPLINE stands for REengineering Software Product Variants into Software Product LINE

❶ Chapter 2 gives a background about software product line and software variants. It also presents the techniques used in our approach supported by illustrative examples (*i.e.* Formal Concept Analysis, Relational Concept Analysis and Latent Semantic Indexing).

❷ Chapter 3 presents the state of the art regarding our approach. It explains and compares the feature location approaches (single software and software family), source code documentation (*i.e.* code comprehension) approaches and approaches that extract feature models from different artifacts. This chapter studies the related work, lists the different used technologies, and puts the light on current issues. Some approaches are described in detail. Finally, when concluding this chapter, we point on weaknesses and strengths (*resp.* comparing with our approach) of these approaches.

❸ Chapter 4 presents the first contribution of our proposal on feature location in a collection of software product variants. In this chapter, we mine functional features from the source code of software product variants as a set of source code elements. We exploit the commonality and variability across software variants to reduce search space. Then, we rely on both lexical (*i.e.* textual) and structural (*i.e.* code dependencies) similarity between source code elements to mine feature implementations.

❹ Chapter 5 describes the feature documentation process (*i.e.* the second contribution). We document the mined feature implementations using two ways. Firstly, we rely on the use-case diagrams of software product variants to document the mined feature implementation using use-case names and descriptions. In the second way, we rely on the source code elements themselves to document the mined feature implementations.

❺ Chapter 6 describes the process of feature model extraction from the source code of software product variants. We rely on the mined and documented feature from the source code of the software variants to extract all relations and dependencies between these features. The process is based on the *product-by-feature matrix* which represents the valid configuration for each software.

❻ Chapter 7 presents the set of experiments that were conducted using real software variants and software product lines to validate our approach. We present each experiment on two parts: a use case part, where we present the use case that we used for conducting the experiment; and a validation part, where we show and discuss the obtained results.

❽ Chapter 8 draws some conclusions about the proposed approach and describes several future research directions.

❼ Appendix A presents the prototype implementation in Java starting with the structural view of the architecture of REVPLINE as a component diagram. We then explain the role of each component in the proposed approach.

Part I

Background and State of the Art

CHAPTER 2

BACKGROUND

Learn from science that you must doubt the experts. As a matter of fact, I can also define science another way: Science is the belief in the ignorance of experts.

Richard P. FEYNMAN

Preamble

In this chapter, we present the background needed to understand our proposal. Section 2.1 presents the main concepts of Software Product Line Engineering and software product variants. Section 2.2 presents the Formal Concept Analysis classification technique, as well as its extension called the Relational Concept Analysis. We give the basic formal definitions for these two techniques, supported with illustrative examples. Finally, in Section 2.3 we present the Latent Semantic Indexing technique, supported with illustrative examples.

2.1 Software Product Line and Software Variants

Th idea of Software Product Line approach is to manage a family of similar or related software products in a reuse-based way. Reuse helps to reduce development and maintenance effort, shorten time-to-market and improve overall quality of software. In this section, we present the important issues regarding Software Product Line Engineering and software product variants.

2.1.1 Software Product Line Engineering

The traditional focus of software engineering is to develop single software (one software system at a time). A typical development process begins with the analysis of customers' requirements and then several development steps are performed (specification, design, implementation, testing). The result obtained is a single software product. Software Product Line Engineering (SPLE) focuses on the development of multiple similar software systems from common core assets [Clements and Northrop, 2002] [Pohl *et al.*, 2005].

Definition 2.1. *"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" [Clements and Northrop, 2002].*

Reuse is one of the major goals of SPLE for its potential to reduce cost and time of software development. With increasing count of lines of code the need for reuse grows. As the number of features and the number of product variants grows, it is worth re-engineering software variants into a SPL for systematic reuse [Xue *et al.*, 2012].

Definition 2.2. *Software reuse is the process of creating software systems from existing software rather than building software systems from scratch [Krueger, 1992].*

2.1.1.1 Motivations for SPLE

Firms are able to reduce their costs and time to market by a factor of 10 or more if they use a SPLE approach [Schmid and Verlage, 2002]. Figure 2.1 is taken from [Clements and Northrop, 2002], it compares the traditional development approach (*i.e.* developing the individual product variants independently) with a SPL approach. It shows the accumulated costs needed to develop *n* different systems. The solid line sketches the costs of developing the systems independently, while the dashed line shows the costs for SPLE. It shows that the initial costs of a SPL might be higher than the one of the traditional approach, but it also shows that the more products there are in the product line, the cheaper a single product becomes. The cumulative cost of a SPLE approach is cheaper than the traditional approach if there are more than approximately three products [McGregor *et al.*, 2002].

The main motivations for creating a SPL are [Pohl *et al.*, 2005] [Clements and Northrop, 2002]: reduction of development costs and time, enhancement of quality, reduction of time to market, reduction of maintenance effort, coping with evolution and complexity, improving cost estimation and increasing customer satisfaction and decreasing labor cost.

SPLE relies on the idea of mass customization known from many industries [Pine, 1993]. Mass customization takes advantage of similarity principle and modular design to massively produce customized products. Many industries are building the same multiple similar software-intensive products over and over again. Therefore, there is an opportunity to massively reuse common software artifacts. Software mass customization focuses on the means of efficiently producing and maintaining multiple similar software products, exploiting what they have in common and managing what varies among them. SPLE aims at developing related variants in a systematic way and providing appropriate solutions for different customers [Clements and Northrop, 2002]. Instead of individually developing each variant from scratch, commonalities are considered only once.

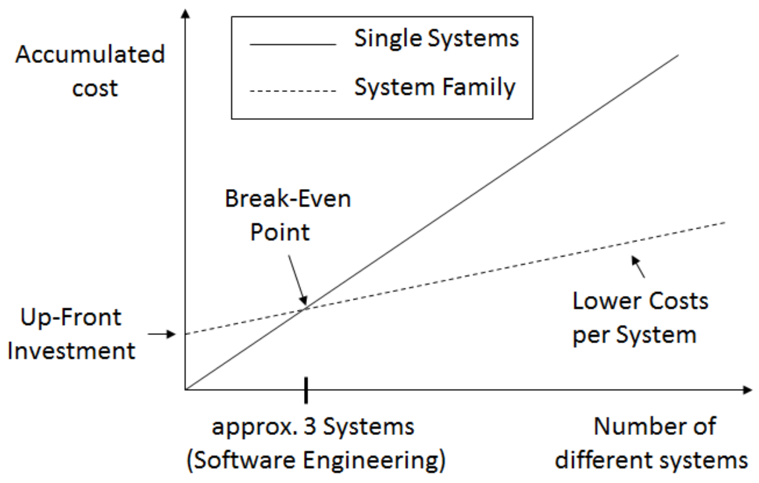


Figure 2.1 : Costs for developing *n* kinds of systems as single systems compared to SPLE [Clements and Northrop, 2002].

2.1.1.2 Software Product Line Engineering Process

SPLE essentially is a two-phase approach which consists of domain engineering and application engineering (*cf.* Figure 2.2 [Apel *et al.*, 2013]). Application domain is a software area, which contains the common parts among the similar software products. The mission of domain engineering is to build the SPL architecture consisting of a core asset and the software variant features, while the application engineering focus on the derivation of the new products by the different customizations of software variant features applied onto the core asset. We present here the engineering process for software product lines. Namely domain engineering and application engineering.

❶ Domain Engineering. A SPL must fulfil not only the requirements of a single customer but the requirements of multiple customers in a domain, including both current customers and potential future customers. Thus, the entire domain and its potential requirements are analyzed. The process to develop a set of related software products instead of single software is called domain engineering. Domain Engineering is development for reuse; common and variable artifacts (requirements, source code, components, test cases, *etc.*) are factored so that their reuse is enabled. During domain analysis, the commonalities and differences between potential software variants are identified and described (*e.g.* in terms of features). Then, developers design and implement the SPL such that different software variants can be constructed from common and variable parts [Apel *et al.*, 2013] [Clements and Northrop, 2002]. Domain engineering consists of four activities: domain analysis, domain design, domain realization (coding) and domain testing [van der Linden *et al.*, 2007].

❷ Application Engineering. Application engineering is the process of deriving a single variant tailored to the requirements of a specific customer from a software product line, based on the results of domain engineering. Application engineering is a development with reuse. The process of creating software products from the domain assets is called product derivation. Real software products are derived using the common and reusable artifacts developed in domain engineering. Application engineering is composed of four activities (keeping with activities of domain engineering): application requirements engineering, application design, application coding, and application testing [Apel *et al.*, 2013]. This process is built on the domain engineering process and consists in developing a final product, by reusing the reusable artifacts (*e.g.* source code) and adapting the final product to specific requirements. Ideally,

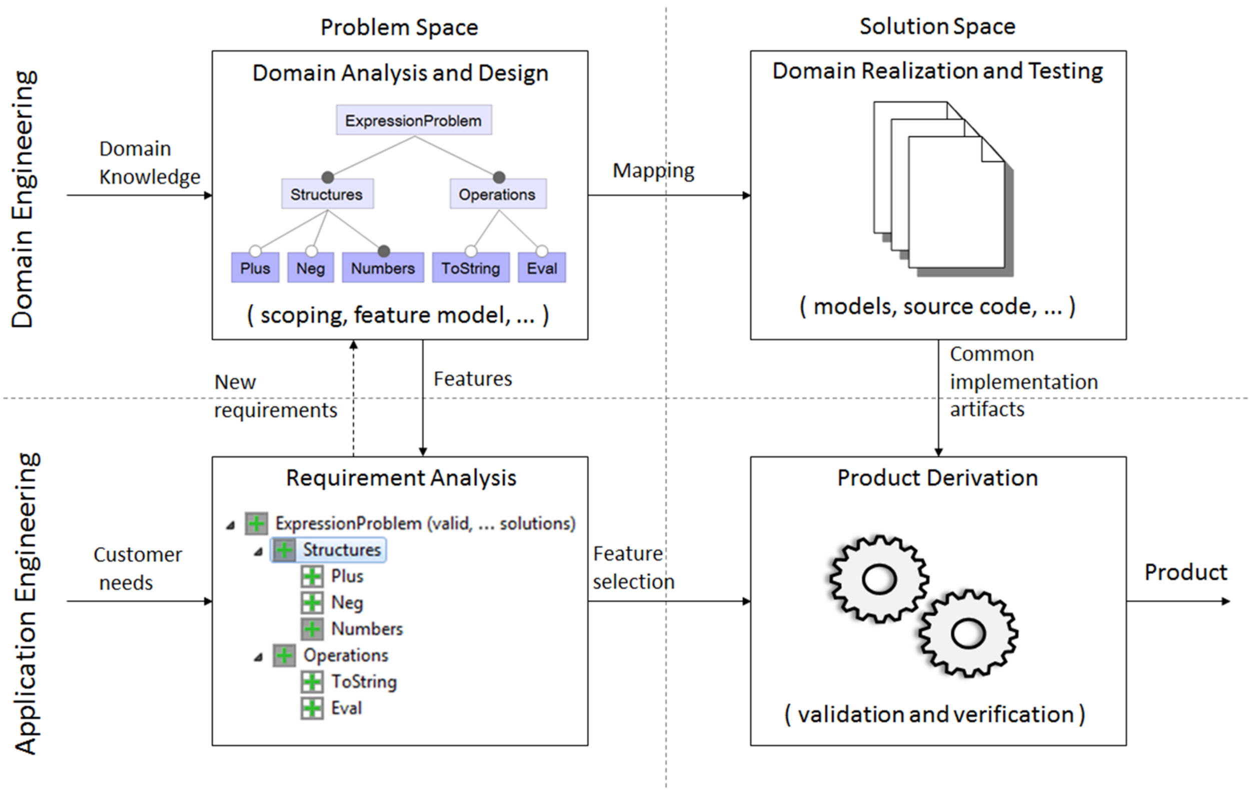


Figure 2.2 : Overview of an engineering process for software product lines [Apel *et al.*, 2013].

the customer's requirements can be mapped to elements (*e.g.* features) identified through domain engineering, so that the software variant can be built from existing common and variable parts of the SPL implementation [Clements and Northrop, 2002]. Depending on the form of implementation, there can be different automation levels of the application engineering process, from manual development effort to more advanced technology including automated variant configuration and generation [van der Linden *et al.*, 2007].

2.1.1.3 Variability Management

Central and unique to SPLE is the management of variability, *i.e.* the process of factoring out commonalities and systematizing variabilities of requirements, source code, documentation, test artifacts, models. It is one of the fundamental principles to successful SPLE. Variability management includes the activities of clearly representing variability in software artifacts throughout the life cycle, managing dependencies among different variabilities, and supporting the instantiations of those variabilities. It includes extremely complex and challenging tasks, which needs to be supported by appropriate approaches, techniques, and tools. Systematically identifying and appropriately managing variabilities across different software products of a family are the key characteristics that distinguish SPLE from other reuse-based software development approaches [Chen *et al.*, 2009]. Several definitions of variability have been given in the literature:

- Weiss and Lai [Weiss and Lai, 1999] define variability in SPL as "an assumption about how members of a family may differ from each other".

- Svahnberg *et al.* [Svahnberg *et al.*, 2005] define variability as "the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context".

❒ Pohl *et al.* [Pohl *et al.*, 2005] define variability in time as "the existence of different versions of an artifact that are valid at different times" and the variability in space as "the existence of an artifact in different shapes at the same time".

2.1.1.4 Variability categories, levels, and classification

Variability may be identified from different viewpoints. Bachmann and Bass [Bachmann and Bass, 2001] classify variabilities into categories: variability in function (a particular function may exist in some products and not in others), variability in control flow means (a particular interaction may occur in some products and not in others), variability in data (a particular data structure may be used in one product but not in another), variability in technology (hardware, OS, user interface, *etc.* may differ from one product to another), variability in product quality goals means that goals (qualitative dimensions like security or performance may be unique to one product), variability in space (variability in space covers the simultaneous use of a variable artefact in different shapes of different products such as keypads, magnetic cards, and fingerprint scanners of the home automation system) [Clements and Northrop, 2002].

Svahnberg *et al.* [Svahnberg *et al.*, 2005] have defined five levels of variability (SPL, product, component, sub-component and code level) where different variability design issues appear. Extra classification, based on the difference between essential and technical variability, has been proposed in [Halmans and Pohl, 2004]. Essential variability refers to the customer's point of view (and is also called external variability [Pohl *et al.*, 2005] or product line variability [Metzger *et al.*, 2007]). Technical variability (also called internal variability [Pohl *et al.*, 2005] or software variability [Metzger *et al.*, 2007]) refers to the SPL engineer's point of view who is mainly interested in implementation details.

Feature model is presently the most popular technique to model variability [Clements and Northrop, 2002]. Many extensions and dialects of feature models have been proposed in literature (*e.g.* FODA [Kang, 1990], FORM [Kang *et al.*, 1998], and FeatureRSEB [Griss *et al.*, 1998]) (*cf.* Section 2.1.1.6).

2.1.1.5 Feature and Variability

Feature models aim at describing the variability of a SPL in terms of features. Due to the diversity of software engineering research, there are several definitions of a features, for example:

❏ [Apel *et al.*, 2013]: "A feature is a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle".

❏ [Apel and Kästner, 2009]: "a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option".

❏ [Kang, 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 and Eisenecker, 2000]: "a distinguishable characteristic of a concept (*e.g.* system, component, and so on) that is relevant to some stakeholder of the concept".

❏ [Czarnecki and Eisenecker, 2000]: "anything users or client programs might want to control about a concept".

❏ [Zave and Jackson, 1997]: "an optional or incremental unit of functionality".

- [Batory, 2005]: "an increment of program functionality".
- [Bosch, 2000]: "a logical unit of behavior specified by a set of functional and non-functional requirements".
- [Chen *et al.*, 2005]: "a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements".
- [Batory *et al.*, 2003]: "a product characteristic that is used in distinguishing programs within a family of related programs".
- [Classen *et al.*, 2008]: "a triplet, f = (R, W, S), where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification".

We consider that Kang [Kang, 1990] definition is the closest definition to our work. In our work, we focus on functional features. Functional features express the behavior or the way users may interact with a software system or systems [Al-Msie'deen *et al.*, 2013b].

2.1.1.6 Feature Models

The terms Feature Model (FM) and feature diagram are employed in the literature, usually to denote the same thing. The important aspect of FMs is the modelling of variability. The feature diagram notation was first introduced by Kang *et al.* [Kang, 1990] in their Feature Oriented Domain Analysis (FODA) methodology. FODA is a method to represent variability and commonality among related software systems. Several definitions of FM have been given in the literature:

- Czarnecki *et al.* [Czarnecki *et al.*, 2006] define FM as a tree-like hierarchy of features and constraints between them (a FM is a hierarchy of features with variability).
- Kang [Kang, 1990] defines FM as a *de facto* standard to model the common and variable features of a SPL and their relationships.

FMs were first introduced as a part of the FODA method. Since then, feature modelling has been widely adopted by the SPL community. However, many extensions to the original proposal have been presented and a consensus about a FM notation has not been reached yet.

FeatuRSEB [Griss *et al.*, 1998], FORM [Kang *et al.*, 1998] or the graphical notation proposed in [Czarnecki and Eisenecker, 2000] are only slightly different from the FODA notation (boxes are added around feature names or filled circles are used to represent mandatory features). In addition, a number of textual FM languages were also proposed in the literature. The first textual language was Feature Description Language (FDL) [van Deursen and Klint, 2002].

A FM defines which feature combinations lead to valid products within the SPL. The individual features are depicted as labelled boxes and are arranged in a tree-like structure. There is always exactly one root feature that is included in every valid program configuration. Each feature, apart from root, has a single parent feature and every feature can have a set of child features. These child-parent relationships are denoted via connecting lines. Notice here that a child feature can only be included in a program configuration if its parent is included as well. There are four different kinds of relations in which a child (*resp.* a set of children) can interrelate with its parent [Haslinger *et al.*, 2011]:

- If a feature is optional (depicted with an empty circle at the child end of the relation) it may or may not be selected if its parent feature is selected.

- If a feature is mandatory (depicted with a filled circle at the child end of the relation) it has to be selected whenever its parent feature is selected.
- If a set of features forms an inclusive-or relation (depicted as filled arcs) at least one feature of the set has to be selected if its parent is selected (*i.e.* or relation).
- If a set of features forms an exclusive-or relation (depicted as empty arcs) exactly one feature of the set has to be selected if its parent is selected (*i.e.* alternative relation).

Besides the child-parent relations there are also so called cross-tree constraints (CTCs). They capture any arbitrary relation among features, usually denoted by propositional logic formulas [Benavides *et al.*, 2010]. The most common CTCs are the require and exclude relations. If feature A require feature B, then feature B has to be included whenever feature A is included. If two features are in an exclude relation then these two features cannot be selected together in any valid product configuration. FMs that do not include any CTC are referred to as basic FMs.

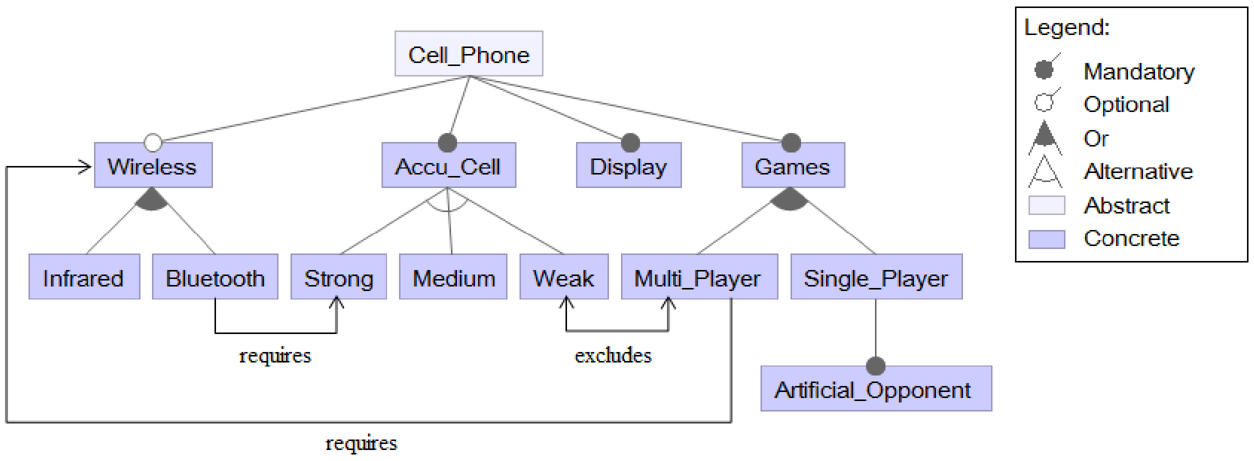


Figure 2.3 : Cell phone SPL feature model.

Figure 2.3 shows the feature model of the *cell phone* SPL [Haslinger, 2012]. Feature *Cell_Phone* is the root feature of this FM; hence it is selected in every program configuration. It has three mandatory child features (*i.e.* the features *Accu_Cell*, *Display* and *Games*), which are also selected in every product configuration as their parent is always included. The children of feature *Accu_Cell* form an *exclusive-or* relation, meaning that the programs of this SPL include exactly one of the features *Strong*, *Medium* or *Weak*. The features *Multi_Player* and *Single_Player* constitute an *inclusive-or*, which necessitates that at least one of these two features is selected in any valid program configuration. *Single_Player* has *Artificial_Opponent* as a mandatory child feature. Feature *Wireless* is an optional child feature of root; hence it may or may not be selected. Its child features *Infrared* and *Bluetooth* form an *inclusive-or* relation, meaning that if a program includes feature *Wireless* then at least one of its two child features has to be selected as well.

The cell phone SPL also introduces three CTCs. While feature *Multi_Player* cannot be selected together with feature *Weak*, it cannot be selected without feature *Wireless*. Lastly, feature *Bluetooth* require feature *Strong*.

```
P1 = {Cell_Phone, Accu_Cell, Strong, Display, Games, Single_Player, Artificial_Opponent}
```

LISTING 2.1 : Valid product configuration.

Listing 2.2 represents an illegal program configuration according to the FM shown in Figure 2.3, as features Strong and Medium are both selected which is prohibited by the FM as these two features are in

```
P2 = {Cell_Phone, Accu_Cell, Strong, Medium, Display, Games, Multi_Player}
```

LISTING 2.2 : Illegal product configuration.

an exclusive-or relation. Listing 2.1 represents a valid program configuration according to the FM shown in Figure 2.3. Table 2.1 shows the 16 valid feature sets (*i.e.* product configurations) defined by the FM in Figure 2.3.

Table 2.1 : Valid product configurations of cell phone SPL.

	Cell_Phone	Wireless	Infrared	Bluetooth	Accu_Cell	Strong	Medium	Weak	Display	Games	Multi_Player	Single_Player	Artificial_Opponent
Product-1	✗				✗	✗			✗	✗		✗	✗
Product-2	✗				✗		✗		✗	✗		✗	✗
Product-3	✗				✗			✗	✗	✗		✗	✗
Product-4	✗	✗		✗	✗	✗			✗	✗		✗	✗
Product-5	✗	✗		✗	✗	✗			✗	✗	✗		
Product-6	✗	✗		✗	✗	✗			✗	✗	✗	✗	✗
Product-7	✗	✗	✗		✗	✗			✗	✗		✗	✗
Product-8	✗	✗	✗		✗	✗			✗	✗	✗		
Product-9	✗	✗	✗		✗	✗			✗	✗	✗	✗	✗
Product-10	✗	✗	✗		✗		✗		✗	✗		✗	✗
Product-11	✗	✗	✗		✗		✗		✗	✗	✗		
Product-12	✗	✗	✗		✗		✗		✗	✗	✗	✗	✗
Product-13	✗	✗	✗		✗			✗	✗	✗		✗	✗
Product-14	✗	✗	✗	✗	✗	✗			✗	✗		✗	✗
Product-15	✗	✗	✗	✗	✗	✗			✗	✗	✗		
Product-16	✗	✗	✗	✗	✗	✗			✗	✗	✗	✗	✗

The primary purpose of a hierarchy in FM is to organize a potentially large number of features into multiple levels of increasing detail. The hierarchy is usually represented as a rooted tree, the root feature being the most general concept. Usually, the root feature refers to the product family name [Apel *et al.*, 2013]. FMs can be used in different phases of the SPL development, from high-level requirements to code implementation. From an initial stage (*e.g.* requirements elicitation) to components and platform modelling, FMs can be applied to any kind of artifacts (*e.g.* source code, documentation, models, *etc.*) and at any level of abstraction [Clements and Northrop, 2002].

2.1.2 Software Product Variants

Software product variants are a collection of similar software variants which share some artifacts (*e.g.* source code, requirement, *etc.*) and differ in other ones, to accommodate specific demands of customers in a particular domain [Ye *et al.*, 2009] [Xue *et al.*, 2012]. Software product variants often evolve from first software developed for and successfully used by the first customer. Wingsoft Financial Management Systems [Ye *et al.*, 2009], Mobile Media, and Linux kernel [Xue *et al.*, 2012] are some of the many examples of such product evolution. These software product variants generally share some common features (source code) but they are also different from one another due to subsequent customization to meet specific requirements of different stakeholders.

Usually software product variants, developed by *clone-and-own* approach, form often a starting point for building a SPL. The clone-and-own approach refers to the practice of creating a new software product by "cloning" a copy of the source code of another software product and then modifying it, thereby taking ownership of the newly cloned software which now has a development and maintenance life cycle of its own [Rubin and Chechik, 2013a] [Dubinsky *et al.*, 2013]. Existing software products, assets (including requirements, architecture, source code, and so forth), decision models, and production mechanisms can often be reused and re-engineered when a SPL approach is established in order to save time and effort [Rubin and Chechik, 2012]. Figure 2.4 shows an example of two new products (product *Q* and product *R*) being created from the assets of product *P*.

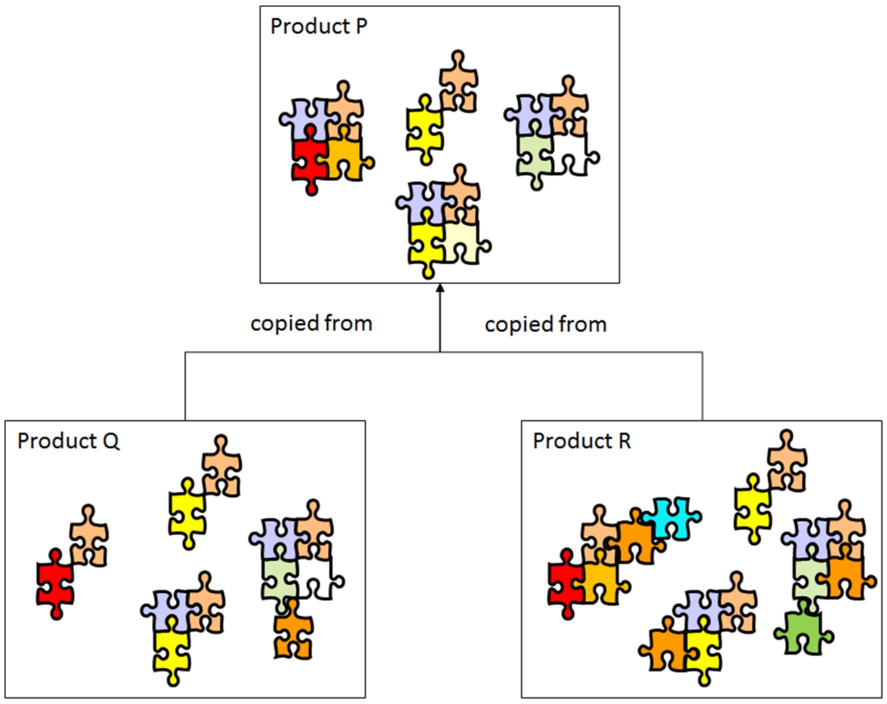


Figure 2.4 : The clone-and-own approach.

Feature location techniques aim at locating software artifacts that implement specific program functionality, *a.k.a.* a feature. These techniques support developers during various activities such as software maintenance. In reality, software variants often emerge *ad-hoc*. Software developers often create new products by using one or more of the available technology-driven software reuse techniques such as duplication (the "clone-and-own" approach), source control branching and preprocessor directives. Identification of the relationship between the features and their corresponding implementation in the source code is the main goal of feature location techniques [Rubin and Chechik, 2013b]. The problem of feature extraction is known under several names in software engineering research: feature mining [She *et al.*, 2011], feature identification [Ziadi *et al.*, 2012], feature location [Xue *et al.*, 2012], fact extraction [Basten and Klint, 2009] and topic mining [Kuhn *et al.*, 2007].

In software systems, a feature represents a functionality that is well-defined by requirements and accessible to developers and users. Software maintenance and evolution involves adding new features to programs, refining existing functionalities, and removing bugs, which is analogous to removing unwanted functionalities. Feature location is one of the most important and common activities performed by developers during software maintenance [Dit *et al.*, 2013], because no maintenance task can be completed without first locating and understanding the code that is relevant to the task at hand [Xue *et al.*, 2012].

2.2 Formal and Relational Concept Analysis

Formal Concept Analysis (FCA) is a classification technique that takes data sets of objects and their attributes, and extracts relations between these objects according to the attributes they share [Ganter and Wille, 1997]. It has many applications in software engineering: [Tilley *et al.*, 2005] [Cellier *et al.*, 2008] [Bhatti *et al.*, 2012]. Relational Concept Analysis (RCA) is an iterative version of FCA in which, the objects are classified not only according to the attributes they share, but also according to the relations between these objects [Huchard *et al.*, 2007]. In our work, we use FCA and RCA as clustering techniques to reduce the search space for feature location, feature documentation and FM extraction.

2.2.1 Formal Concept Analysis

Galois lattices [Barbut and Monjardet, 1970] and concept lattices [Ganter and Wille, 1997] are core structures of a data analysis framework (Formal Concept Analysis, or FCA for short) for extracting an ordered set of concepts from a dataset, called a Formal Context, composed of objects described by attributes. We explain the FCA technique along with a case study about animals and their characteristics. We consider the dataset of Table 2.2. This Table presents a dataset where several animals are described by their characteristics.

Table 2.2 : Animals and their characteristics.

Animal	Characteristics
flying squirrel	flying, with_membrane
bat	flying, nocturnal, with_membrane
ostrich	feathered
flamingo	flying, feathered, migratory
chicken	flying, feathered, with_crest

Definition 2.3. *A formal context is a triple $K = (O, A, R)$ where O and A are sets (objects and attributes, respectively) and R is a binary relation, (i.e. $R \subseteq O \times A$).*

A formal context can be graphically represented as a cross table in which, objects appear as row labels and attributes as column labels. A cross in the cell (o, a) of this table indicates that the object o has the attribute a.

From our case study, we can build a formal context of animals O = {flying squirrel, bat, ostrich, flamingo, chicken} and their characteristics A = {flying, nocturnal, feathered, migratory, with_crest, with_membrane} (*cf.* Table 2.3).

Table 2.3 : A formal context for animals and their characteristics.

	flying	nocturnal	feathered	migratory	with_crest	with_membrane
flying squirrel	✗					✗
bat	✗	✗				✗
ostrich			✗			
flamingo	✗		✗	✗		
chicken	✗		✗		✗	

Definition 2.4. *A formal concept is a pair (E, I) composed of an object set $E \subseteq O$ and their shared attribute set $I \subseteq A$. $E = \{o \in O | \forall a \in I, (o, a) \in R\}$ is the extent of the concept, while $I = \{a \in A | \forall o \in E, (o, a) \in R\}$ is the intent of the concept.*

☞ For example, concept 7 ({*flamingo, chicken*} {*flying, feathered*}) in Figure 2.5 is a concept of our example.

Definition 2.5. *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 $E_1 \subseteq E_2$ (or 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, concept 7 ({flamingo, chicken} {flying, feathered}) in Figure 2.5 is a *sub-concept* of ({flamingo, chicken, ostrich} {feathered}).

Definition 2.6. *Let $\mathscr{C}_K$ be the set of all concepts of a formal context K. This set of concepts provided with the specialization order $(\mathscr{C}_K, \leq_s)$ has a lattice structure, and is called the concept lattice associated with K. It is denoted $\mathscr{C}(O, A, R)$.*

In a concept lattice (*cf.* Figure 2.5) (built using Galicia [Valtchev *et al.*, 2005]), labels can be simplified by putting down each object and each attribute only once. In this way, a lattice with reduced labels (simplified intents and extents) can be read in the same way without losing any information such as Figure 2.6, which is built using the Con Exp[1] (Concept Explorer) [Yevtushenko *et al.*, 2006]. In Figure 2.5 and 2.6, only the Hasse diagram of lattice (edges of the transitive reduction) is shown.

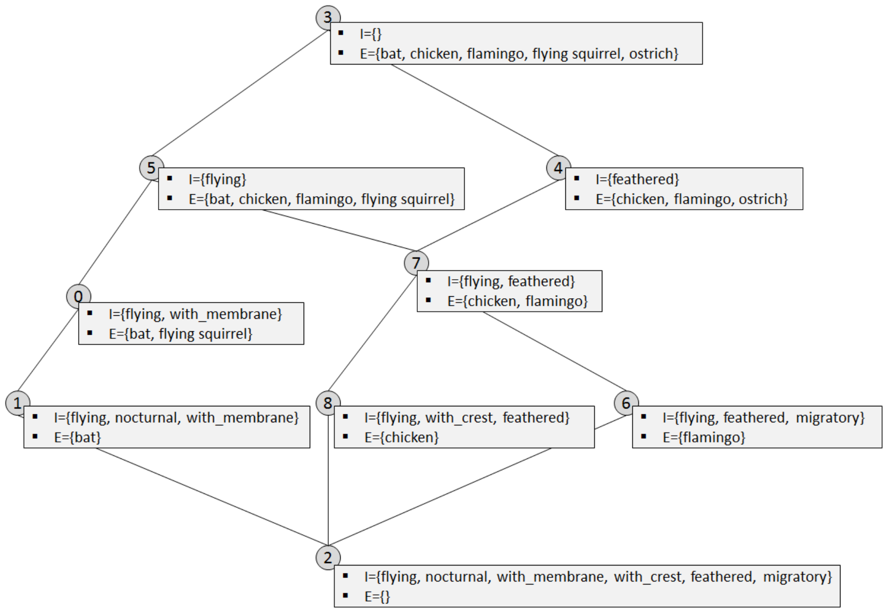

3
I={}
E={bat, chicken, flamingo, flying squirrel, ostrich}
5
I={flying}
E={bat, chicken, flamingo, flying squirrel}
4
I={feathered}
E={chicken, flamingo, ostrich}
7
I={flying, feathered}
E={chicken, flamingo}
0
I={flying, with_membrane}
E={bat, flying squirrel}
1
I={flying, nocturnal, with_membrane}
E={bat}
8
I={flying, with_crest, feathered}
E={chicken}
6
I={flying, feathered, migratory}
E={flamingo}
2
I={flying, nocturnal, with_membrane, with_crest, feathered, migratory}
E={}

Figure 2.5 : The concept lattice for the context in Table 2.3 via Galicia.

Figure 2.7 shows the another view (built with eRCA[2]) of the concept lattice structuring our animals. In this diagram, extents and intents are presented in a simplified form: removing up-down inherited attributes and down-up inherited objects.

The reader may have noticed that, applying the simplification of extents and intents, some concepts, are represented having empty simplified extent and intent. These concepts introduce neither objects nor attributes. In several FCA applications, they can be ignored (*e.g.* in [Godin and Mili, 1993 ; Loesch and Ploedereder, 2007 ; Ryssel *et al.*, 2011]). Reversely, the term *object concept* (*resp. attribute concept*) refers to a concept which introduces at least one object (*resp.* attribute).

[1] http://conexp.sourceforge.net/
[2] http://code.google.com/p/erca/

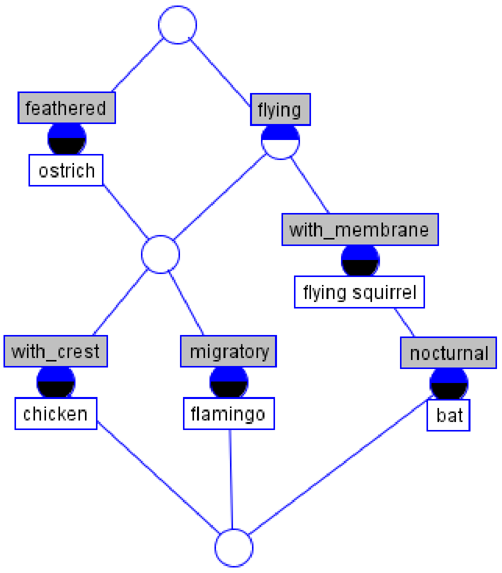


Figure 2.6 : The concept lattice for the context in Table 2.3 via Con Exp.

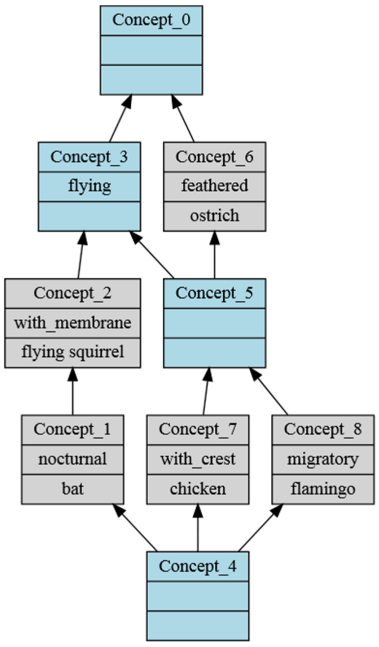


Figure 2.7 : The concept lattice for the formal context of Table 2.3 via eRCA.

Definition 2.7. *The Galois Sub-Hierarchy (GSH) also called AOC-poset (for Attribute-Object-Concept partially ordered set) of a concept lattice R is the sub-order of R induced by the sets of attribute concepts and of object concepts.*

In our approach, we will consider the AOC-poset. For our example, it would correspond to the concept lattice of Figure 2.7 deprived of `Concept_0`, `Concept_4` and `Concept_5` (*cf.* Figure 2.8). Figures 2.7 and 2.8 built using eRCA [Falleri and Dolques, 2010]. In Figure 2.8, the concepts have been renamed by the construction tool (*e.g.* concept_3 of the concept lattice (from Figure 2.7) is concept_5 of the AOC-poset (in Figure 2.8).

There is a drastic difference of complexity between the two structures (the concept lattice

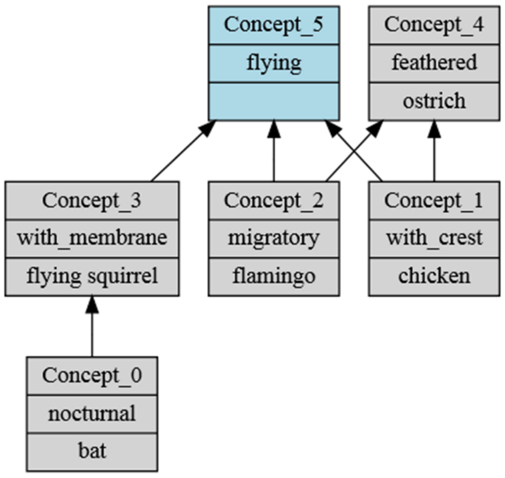


Figure 2.8 : The AOC-poset for the formal context of Table 2.3 via eRCA.

and AOC-poset), because the concept lattice may have $2^{min(|O|,|A|)}$ concepts, while the number of concepts in the AOC-poset is bounded by $|O| + |A|$. Algorithms for building AOC-posets are introduced in [Arévalo *et al.*, 2007] and [Berry *et al.*, 2012].

2.2.2 Relational Concept Analysis

RCA [Huchard *et al.*, 2007] is an iterative version of FCA in which, the objects are classified not only according to the attributes they share, but also according to the relations between them. Other close approaches are: [Prediger and Wille, 1999] [Ferré *et al.*, 2005] [Baader and Distel, 2008]. Let us take the following example (the Mexican food example) [Azmeh *et al.*, 2011]. In this case study there is a list of countries, a list of restaurants, a list of Mexican dishes, a list of ingredients, and finally a list of salsas. There are some relations between these entities (Country, Restaurant, MexicanDish, Ingredient, Salsa), such that: a Country "has" a Restaurant; a Restaurant "serves" a MexicanDish; a MexicanDish "contains" an Ingredient; an Ingredient is "made-in" a Country; and finally a Salsa is "suitable-with" a MexicanDish. These entities and their relations are expressed by the directed cyclic graph (the graph contains a cycle) in Figure 2.9. The relational context family (RCF) (*cf.* Table 2.5) is an instance of this entity-relationship diagram.

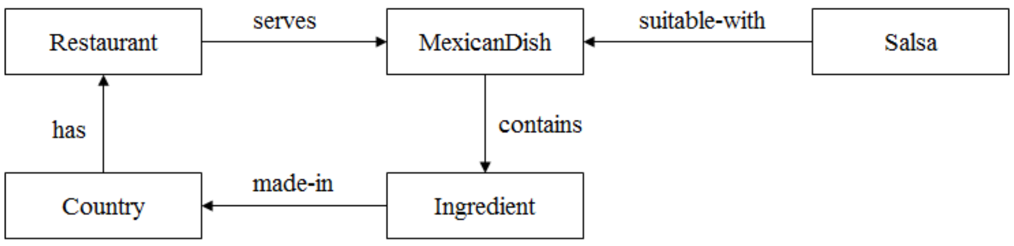


Figure 2.9 : The entities of the Mexican food example.

Definition 2.8. *A relational context family (RCF) is a pair (K, R) where K is a set of formal (object-attribute) contexts $K_i = (O_i, A_i, I_i)$ and R is a set of relational (object-object) contexts $r_{ij} \subseteq O_i \times O_j$, where O_i (domain of r_{ij}) and O_j (range of r_{ij}) are the object sets of the contexts K_i and K_j, respectively.*

The RCF corresponding to the Mexican food example consists of four formal contexts (Country, Restaurant, MexicanDish, Ingredient, and Salsa), illustrated in Table 2.4; and five relational contexts (contains, has, serves, suitable-with, and made-in), illustrated in Table 2.5.

Table 2.4 : The formal contexts of the Mexican Food RCF.

Country	ca	en	fr	lb	mx	es	us	America	Asia	Europe
Canada	✗							✗		
England		✗								✗
France			✗							✗
Lebanon				✗					✗	
Mexico					✗			✗		
Spain						✗				✗
USA							✗	✗		

Restaurant	r1	r2	r3	r4	r5	r6	r7
Chili's	✗						
Chipotle		✗					
El Sombrero			✗				
Hard Rock				✗			
Mi Casa					✗		
Taco Bell						✗	
Old el Paso						✗	✗

Ingredient	i1	i2	i3	i4	i5	i6	i7	i8	i9	i10	i11	i12
chicken	✗											
beef		✗										
pork			✗									
vegetables				✗								
beans					✗							
rice						✗						
cheese							✗					
guacamole								✗				
sour-cream									✗			
lettuce										✗		
corn-tortilla											✗	
flour-tortilla												✗

MexicanDish	d1	d2	d3	d4	d5	d6
Burritos	✗					
Enchiladas		✗				
Fajitas			✗			
Nachos				✗		
Quesadillas					✗	
Tacos						✗

Salsa	s1	s2	s3	s4	mild	medium-hot	hot
Fresh Tomato	✗				✗		
Roasted Chili-Corn		✗				✗	
Tomatillo-Green Chili			✗			✗	
Tomatillo-Red Chili				✗			✗

Table 2.5 : The relational context family of the Mexican Food RCF.

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

has	Chili's	Chipotle	El Sombrero	Hard Rock	Mi Casa	Taco Bell	Old el Paso
Canada	✗	✗	✗	✗		✗	
England		✗		✗		✗	
France			✗	✗			✗
Lebanon	✗			✗		✗	
Mexico	✗				✗	✗	
Spain				✗		✗	
USA	✗	✗	✗	✗	✗	✗	

serves	Burritos	Enchiladas	Fajitas	Nachos	Quesadillas	Tacos
Chili's			✗		✗	✗
Chipotle	✗					✗
El Sombrero	✗	✗	✗	✗	✗	✗
Hard Rock			✗	✗		
Mi Casa	✗	✗		✗	✗	✗
Taco Bell	✗			✗	✗	✗
Old el Paso						✗

made-in
chicken
beef
pork
vegetables
beans
rice
cheese
guacamole
sour-cream
lettuce
corn-tortilla
flour-tortilla

serves	Burritos	Enchiladas	Fajitas	Nachos	Quesadillas	Tacos
Fresh Tomato	✗	✗	✗	✗	✗	✗
Roasted Chili-Corn	✗			✗		
Tomatillo-Green Chili	✗			✗		
Tomatillo-Red Chili	✗	✗	✗	✗	✗	✗

An RCF is used in an iterative process to generate at each step a set of concept lattices. Firstly, concept lattices are built using the formal contexts only. Then, in the following steps, formal contexts are concatenated with the relational contexts enriched with knowledge obtained at a previous step. This enrichment is based on the notion of scaling operators that produce scaled relations.

The scaling mechanism translates the links between objects into conventional FCA attributes (which we call here relational attributes) and derives a collection of lattices whose concepts are linked by relations (*cf.* Figure 2.11 (built using eRCA)). For example, the existentially scaled relation (which we will use in our work) captures the following information: if an object o_s is linked to another object o_t, then in the scaled relation, o_s will receive relational attributes associated to concepts, which group o_t with other objects. This is used to form new groups, for example the group (See $Concept\_85$) of Restaurant lattice, which serve at least one dish containing sour cream ($Concept\_38$) (such dishes are grouped in $Concept\_73$) (*cf.* Figure 2.10). The steps are repeated until the lattices become stable (*i.e.* when no more new concepts are generated).

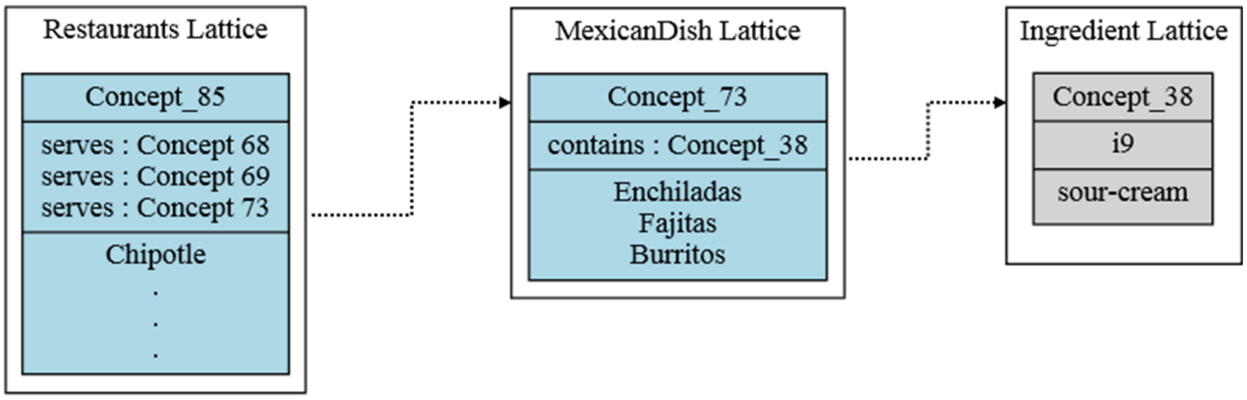


Figure 2.10 : Concept 85, 73 and 38 of the Restaurant, MexicanDish and Ingredient concept lattice.

Definition 2.9. *Let us define* $r_{ij}(o_i) = \{oj \in O_j | (o_i, o_j) \in r_{ij}\}$. *The* ***exists*** *scaled relation* $r^{\exists}_{ij}$ *associated to* $r_{ij} \subseteq O_i \times O_j$ *is defined as* $r^{\exists}_{ij} \subseteq O_i \times \mathscr{C}(O_j)$*, such that:* $(o_i, c) \in r^{\exists}_{ij} \Leftrightarrow \exists x \in r_{ij}(o_i) : x \in Extent(c)$*. Thus,* $\exists$ *is a scaling operator (existential). Let us note that in this definition,* $\mathscr{C}(O_j)$ *is any lattice built on the objects of* O_j*.*

Definition 2.10. *A concept lattice family (CLF) is a set of lattices that correspond to the formal contexts, after enriching them with relational attributes.*

For example, we used the exists scaled relations to generate the concept lattice family corresponding to our case study. It consists of five lattices: Country lattice, Restaurant lattice, MexicanDish lattice, Ingredient lattice and Salsa lattice. Here, we present only three lattices of them in Figure 2.11. The complete concept lattice family is available on our web site[3]. For applying FCA and RCA we used the Eclipse eRCA platform[4]. For more details, the reader is invited to refer to [Hacene *et al.*, 2013].

[3]http://www.lirmm.fr/~seriai/encadrements/theses/rafat//index.php?n=T.CLF
[4]http://code.google.com/p/erca/

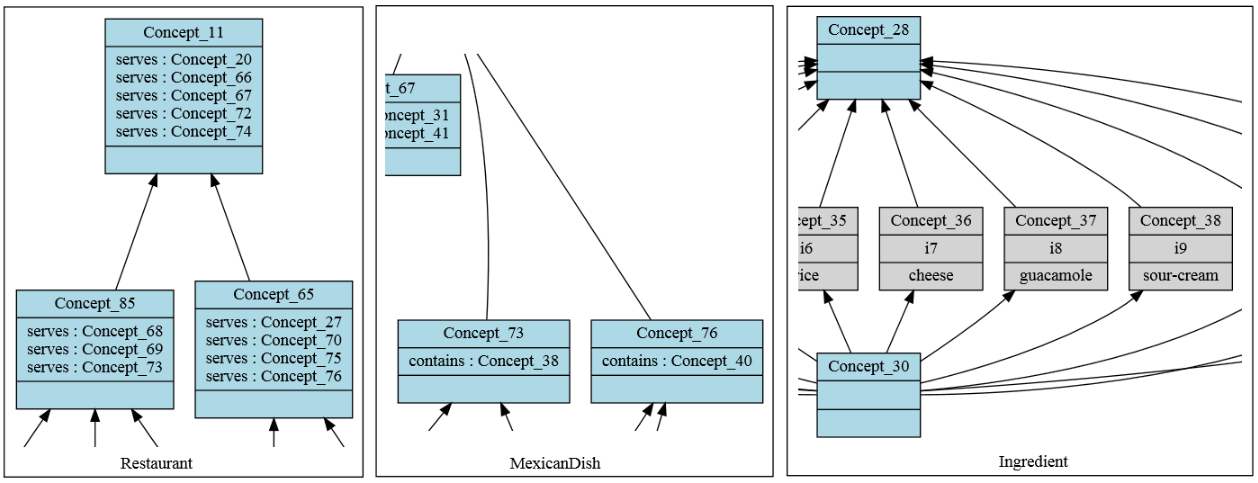


Figure 2.11 : Part of the concept lattice family of the Mexican food example.

2.3 Latent Semantic Indexing

Information Retrieval (IR) refers to techniques that compute textual similarity among different documents. The textual similarity is computed based on the occurrences of terms within documents [Grossman and Frieder, 2004]. If two documents share a large number of terms, those documents are considered to be similar. Different IR techniques have been proposed, such as Latent Semantic Indexing (LSI) and Vector Space Model (VSM), to compute textual similarity[5]. The VSM is the basis for many advanced IR techniques and topic models. The VSM is a simple algebraic model directly based on the *term-document matrix*. The LSI is an IR model that extends the VSM by reducing the dimensionality of the term-document matrix by means of *Singular Value Decomposition* (SVD).

LSI and VSM are examples of IR techniques. In reality, VSM suffers from the two following drawbacks: i) VSM ignores the possible semantic relationship between the terms, and ii) fails to do rank-reduced simplification. LSI became an improvement over the simplistic point of view of term matching, taking into account term dependencies [Deerwester *et al.*, 1990]. LSI assumed that there are some implicit relationships among the words of documents that always appear together even if they do not share any terms; that is to say, there are some latent semantic structures (latent structure) in free text [Cullum and Willoughby, 2002].

In the LSI approach, documents and queries are represented as vectors of terms that occur within documents in a collection [Frakes and Baeza-Yates, 1992]. LSI begins with a term-document matrix, *TDM*, to record the occurrences of the m unique terms within a collection of n documents. In this term-document matrix, each term is represented by a row and each document is represented by a column, with each matrix cell, tdm_{ij}, denoting a measure of the weight of the ith term in the jth document. The weight is actually defined according to the value of term frequency for the ith term in the jth document.

In this thesis, we use the Term Frequency (TF) ($l(i,j)$) [Berry and Browne, 1999] as the local term weight and the Inverse Document Frequency (IDF) ($g(i)$) [Berry and Browne, 1999] as the global weighting functions. The weight w_{ij} is in fact defined according to the value of term frequency for the ith term in the jth document. Exactly, we can write:

[5]For the purpose of our approach, we developed our LSI tool. Available at `https://code.google.com/p/lirmmlsi/`

$$w_{ij} = l(i,j).g(i) \tag{2.1}$$

Where, a local term weight, $l(i,j)$, indicating the relative frequency of the ith term in the jth document, and a global weight, $g(i)$, representing the relative frequency of the ith term within the entire collection of documents. In our work, the most popular weighting is $TFIDF$ (Term Frequency - Inverse Document Frequency) used according to the following equation [Berry and Browne, 1999]:

$$TFIDF_{ij} = (N_{i,j}/N_j^*) \cdot \log(\frac{D}{Di}) \tag{2.2}$$

- where:
  - ❏ $N_{i,j}$ = the number of times word i appears in document j.
  - ❏ N_j^* = the number of total words in document j.
  - ❏ D = the number of documents.
  - ❏ Di = the number of documents in which word i appears.
  - ❏ In Equation 2.1 w_{ij} is $TFIDF_{ij}$, $l(i,j)$ is $(N_{i,j}/N_j^*)$ and $g(i)$ is $\log(\frac{D}{Di})$.

From a geometric point of view, each column of the term-document matrix represents a point in the m-space of the terms. And the whole matrix has n points in this m-space of the terms. Thus, the similarity between two documents can be measured by the cosine of the angle ($\cos\theta$) between the two corresponding points in the m-space of the terms. Commonly, the closer these two points are in the same direction, the more similar these two documents are.

The LSI has the same steps as VSM for the construction of the term-document matrix TDM. LSI is an advanced IR method. The core of LSI is singular value decomposition technique. This technique is used to mitigate noise introduced by stop words (*e.g.* "the", "an", "above") and to overcome two classical problems arising in natural language processing: synonymy and polysemy [Grossman and Frieder, 2004]. Synonymy denotes multiple words that have the same meaning and polysemy means that a single word has multiple meanings (*cf.* Listings 2.3 and 2.4).

Definition 2.11. *Synonymy: two terms $w1$ and $w2$, $w1 \neq w2$, are synonyms if they possess similar semantic meanings.*

```
Beautiful: attractive, pretty, lovely, stunning.
Lucky: auspicious, fortunate.
```

LISTING 2.3 : Two examples of synonymy.

Definition 2.12. *Polysemy: a term w is polysomic if it has multiple semantic meanings.*

```
The farm will fail unless the drought ends soon.
It is difficult to farm this land.
```

LISTING 2.4 : An example of polysemy.

2.3.1 The Singular Value Decomposition Model

The LSI applies Singular Value Decomposition (SVD) to decompose the matrix TDM into the product of three smaller matrices: an $m \times r$ term-concept vector matrix T, an $r \times r$ singular values matrix S, and a $r \times n$ concept-document vector matrix D (r is the rank of matrix TDM). We can represent the relation as follows [Deerwester *et al.*, 1990]:

$$TDM = T_{m\times r} \cdot S_{r\times r} \cdot D_{r\times n} \tag{2.3}$$

LSI allows the optimal approximation for the standard *SVD* by reducing the rank or truncating the singular value matrix S to size $k \ll r$ [Deerwester *et al.*, 1990].

$$TDM \approx TDM_k = T_k \cdot S_k \cdot D_k \tag{2.4}$$

By using the previous two steps of transformations (*i.e.* decomposition and approximation), the initially associated terms are first grouped into a set of concepts at size of r, and then further squeezed into an even smaller set of concepts at size of k. The benefit of such process is to capture most of the important underlying "latent structure" in the association of terms and documents [Cullum and Willoughby, 2002]. The challenges of using LSI are: i) the costly computation of SVD (time complexity), and ii) difficulty in determining the optimal value of k [Xue *et al.*, 2012].

Thus, the choice of value of k is critical to the performance as well as accuracy of LSI. Dumais *et al.* [Dumais, 1992] stated that better results can be obtained when the value of k (number of concepts or topics) is between 235 and 250 for natural language. As the collection of source code files may have a much larger vocabulary, Poshyvanyk *et al.* [Poshyvanyk *et al.*, 2006] reported that the number of topics at 750 produced the best results. But after 750, the result is not stable: some queries got better results and some got worse. Selecting the appropriate number of dimensions for the LSI representation is an open research question.

Similarity between documents in a corpus is described by a cosine similarity matrix whose columns and rows both represent vectors of documents: documents as columns and queries as rows. Similarity is computed as a cosine similarity given by Equation 2.5 [Berry and Browne, 1999], where d_q is a query vector, d_j is a document vector and $W_{i,q}$ and $W_{i,j}$ range over weights of query and document vectors, respectively.

$$cosine\ similarity\,(d_q, d_j) = \frac{\overrightarrow{d_q} \cdot \overrightarrow{d_j}}{|\overrightarrow{d_q}||\overrightarrow{d_j}|} = \frac{\sum_{i=1}^{n} W_{i,q} * W_{i,j}}{\sqrt{\sum_{i=1}^{n} W_{i,q}^2}\sqrt{\sum_{i=1}^{n} W_{i,j}^2}} \tag{2.5}$$

2.3.2 Singular Value Decomposition Numerical Example

We show here, a SVD numerical example called "technical memo example"; this example is taken from [Deerwester *et al.*, 1990]. Table 2.6 presents a sample data set consisting of the titles of 9 technical memoranda. Terms occurring in more than one title are italicized. There are two classes of documents - five about human-computer interaction (c1-c5) and four about graphs (m1-m4).

Table 2.6 : LSI example: technical memo example.

Titles	Text body
c1	Human machine interface for Lab ABC computer applications
c2	A survey of user opinion of computer system response time
c3	The EPS user interface management system
c4	System and human system engineering testing of EPS
c5	Relation of user-perceived response time to error measurement
m1	The generation of random, binary, unordered trees
m2	The intersection graph of paths in trees
m3	Graph minors IV: Widths of trees and well-quasi-ordering
m4	Graph minors: A survey

This dataset can be described by means of a term-document matrix where each cell entry indicates the frequency with which a term occurs in a document [6]. Figure 2.12 part (a) shows a representation of the singular value decomposition for a $m \times n$ matrix of TDM.

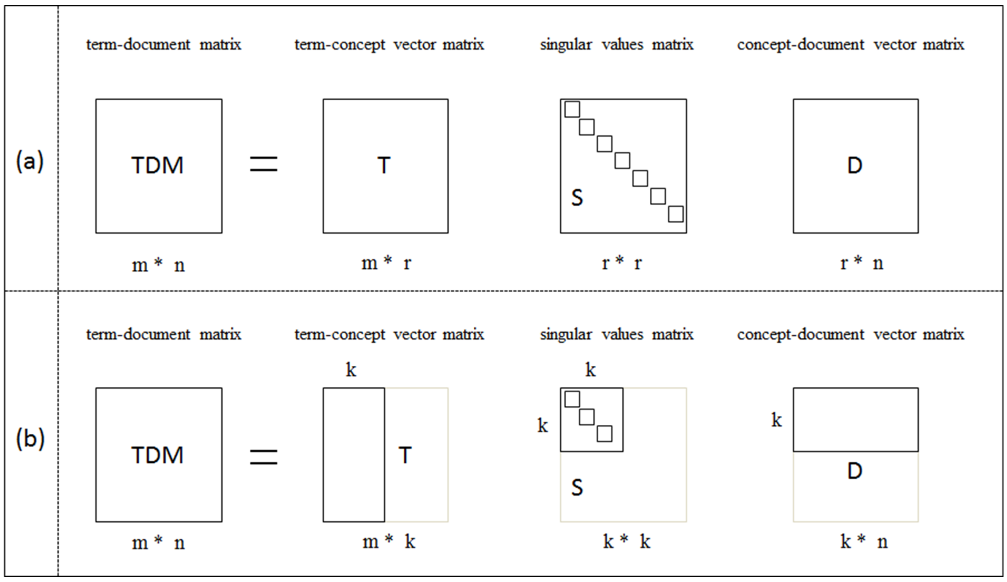


Figure 2.12 : Graphical representation of the SVD (*resp.* reduced SVD) for TDM.

■ Figure 2.12 explains the singular value decomposition of the term-document matrix, TDM. Where [Deerwester *et al.*, 1990]:

- ❑ T has orthogonal, unit-length columns.
- ❑ D has orthogonal, unit-length columns.
- ❑ S is the diagonal matrix of singular values.
- ❑ m is the number of rows of TDM.
- ❑ n is the number of columns of TDM.
- ❑ r is the rank of TDM ($\leq min(t,d)$).

[6]The term-document matrix is available at http://www.lirmm.fr/~seriai/encadrements/theses/rafat/index.php?n=T.TDM

Figure 2.12 part (b) shows the reduced SVD of the term-document matrix, TDM. The notation is the same in the previous figure (Figure 2.12 part (a)) except that $k(\leq m)$ is the chosen number of dimensions (factors) in the reduced model. In this example, we warily chose documents and terms so that SVD would produce an acceptable solution using just two dimensions. Table 2.7 shows the two-dimensional geometric representation for terms and documents that results from the SVD analysis.

Table 2.7 : The 12-term by 9-document matrix.

Terms	Documents								
	c1	c2	c3	c4	c5	m1	m2	m3	m4
human	1	0	0	1	0	0	0	0	0
interface	1	0	1	0	0	0	0	0	0
computer	1	1	0	0	0	0	0	0	0
user	0	1	1	0	1	0	0	0	0
system	0	1	1	2	0	0	0	0	0
response	0	1	0	0	1	0	0	0	0
time	0	1	0	0	1	0	0	0	0
EPS	0	0	1	1	0	0	0	0	0
survey	0	1	0	0	0	0	0	0	1
trees	0	0	0	0	0	1	1	1	0
graph	0	0	0	0	0	0	1	1	1
minors	0	0	0	0	0	0	0	1	1

- Computing the SVD of the TDM matrix presented above (Table 2.7) results in the following three matrices for T, S, D ($TDM = T_{m\times r} \cdot S_{r\times r} \cdot D_{r\times n}$) (rounded to three decimal places) (*cf.* Tables 2.8, 2.9, and 2.10).
  - ❑ T (9-dimensional left-singular vectors for 12 terms).
  - ❑ S (diagonal matrix of 9 singular values).
  - ❑ D (9-dimensional right-singular vectors for 9 documents).

Table 2.8 : SVD – The T (term) matrix.

-0.221	-0.113	0.289	-0.415	-0.106	-0.341	-0.523	0.060	0.407
-0.198	-0.072	0.135	-0.552	0.282	0.496	0.070	0.010	0.109
-0.240	0.043	-0.164	-0.595	-0.107	-0.255	0.302	-0.062	-0.492
-0.404	0.057	-0.338	0.099	0.332	0.385	-0.003	0.000	-0.012
-0.644	-0.167	0.361	0.333	-0.159	-0.207	0.166	-0.034	-0.271
-0.265	0.107	-0.426	0.074	0.080	-0.170	-0.283	0.016	0.054
-0.265	0.107	-0.426	0.074	0.080	-0.170	-0.283	0.016	0.054
-0.301	-0.141	0.330	0.188	0.115	0.272	-0.033	0.019	0.165
-0.206	0.274	-0.178	-0.032	-0.537	0.081	0.467	0.036	0.579
-0.013	0.490	0.231	0.025	0.594	-0.392	0.288	-0.255	0.225
-0.036	0.623	0.223	0.001	-0.068	0.115	-0.160	0.681	-0.232
-0.032	0.451	0.141	-0.009	-0.300	0.277	-0.339	-0.678	-0.183

We now approximate TDM keeping only the first two singular values and the corresponding columns from the T and D matrices (Number of concepts $K = 2$) ($TDM_k = T_k \cdot S_k \cdot D_k$ [Dumais *et al.*, 1988]) (*cf.* Table 2.11).

Multiplying out the matrices TSD' gives the result shown in Table 2.12. This matrix contains only the first k independent linear components of TDM, captures the major associational structure in the matrix and throws out the noise.

Table 2.9 : SVD – The singular values (S) matrix.

3.341	0	0	0	0	0	0	0	0
0	2.542	0	0	0	0	0	0	0
0	0	2.354	0	0	0	0	0	0
0	0	0	1.645	0	0	0	0	0
0	0	0	0	1.505	0	0	0	0
0	0	0	0	0	1.306	0	0	0
0	0	0	0	0	0	0.846	0	0
0	0	0	0	0	0	0	0.560	0
0	0	0	0	0	0	0	0	0.364

Table 2.10 : SVD – The D (document) matrix.

-0.197	-0.606	-0.463	-0.542	-0.279	-0.004	-0.015	-0.024	-0.082
-0.056	0.166	-0.127	-0.232	0.107	0.193	0.438	0.615	0.530
0.110	-0.497	0.208	0.570	-0.505	0.098	0.193	0.253	0.079
-0.950	-0.029	0.042	0.268	0.150	0.015	0.016	0.010	-0.025
0.046	-0.206	0.378	-0.206	0.327	0.395	0.349	0.150	-0.602
-0.077	-0.256	0.724	-0.369	0.035	-0.300	-0.212	0.000	0.362
-0.177	0.433	0.237	-0.265	-0.672	0.341	0.152	-0.249	-0.038
0.014	-0.049	-0.009	0.019	0.058	-0.454	0.762	-0.450	0.070
0.064	-0.243	-0.024	0.084	0.262	0.620	-0.018	-0.520	0.454

Table 2.11 : The $TDM = TSD'(T \cdot S \cdot D)$.

-0.221	-0.113
-0.198	-0.072
-0.240	0.043
-0.404	0.057
-0.644	-0.167
-0.265	0.107
-0.265	0.107
-0.301	-0.141
-0.206	0.274
-0.013	0.490
-0.036	0.623
-0.032	0.451

3.341	0.000
0.000	2.542

-0.197	-0.606	-0.463	-0.542	-0.279	-0.004	-0.015	-0.024	-0.082
-0.056	0.166	-0.127	-0.232	0.107	0.193	0.438	0.615	0.530

Table 2.12 : The TDM_k matrix.

0.161	0.400	0.378	0.467	0.175	-0.052	-0.115	-0.159	-0.092
0.141	0.371	0.330	0.401	0.165	-0.033	-0.070	-0.097	-0.043
0.152	0.504	0.357	0.409	0.235	0.024	0.060	0.086	0.124
0.258	0.842	0.607	0.698	0.392	0.033	0.084	0.122	0.188
0.448	1.234	1.050	1.265	0.555	-0.073	-0.154	-0.210	-0.049
0.159	0.581	0.375	0.417	0.276	0.056	0.132	0.189	0.217
0.159	0.581	0.375	0.417	0.276	0.056	0.132	0.189	0.217
0.218	0.550	0.511	0.628	0.242	-0.065	-0.142	-0.196	-0.107
0.097	0.533	0.230	0.211	0.267	0.137	0.316	0.445	0.426
-0.061	0.233	-0.138	-0.266	0.145	0.241	0.546	0.767	0.664
-0.065	0.336	-0.146	-0.302	0.203	0.306	0.696	0.977	0.849
-0.043	0.255	-0.096	-0.208	0.152	0.222	0.504	0.707	0.616

2.3.3 Evaluation Metrics

The effectiveness of IR methods is usually measured by metrics including *recall*, *precision* and *F-measure*. For a given query, recall is the percentage of correctly retrieved documents to the total number of relevant documents (*cf.* Equation 2.6), while precision is the percentage of correctly retrieved documents to the total number of retrieved documents (*cf.* Equation 2.7). F-Measure defines a trade-off between precision and recall (*cf.* Equation 2.8), so that it gives a high value only in cases where both recall and precision are high [Bissyande *et al.*, 2013].

$$Recall = \frac{|\{relevant\ documents\} \bigcap \{retrieved\ documents\}|}{|\{relevant\ documents\}|} \tag{2.6}$$

$$Precision = \frac{|\{relevant\ documents\} \bigcap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|} \tag{2.7}$$

$$F - Measure = 2 \times \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.8}$$

All measures have values in [0, 1]. If recall equals 1, all relevant documents are retrieved. However, some retrieved documents might not be relevant. If precision equals 1, all retrieved documents are relevant. Nevertheless, relevant documents might not be retrieved. If F-Measure equals 1, all relevant documents are retrieved. However, some retrieved documents might not be relevant.

2.3.4 Latent Semantic Indexing Through Example

Let us take the following example (taken from [Grossman and Frieder, 2004]) to illustrate how LSI works[7]. In this example, the corpus consists of three documents and one query. The documents, query, and k-topics are:

- $d1$: Shipment of gold damaged in a fire.
- $d2$: Delivery of silver arrived in a silver truck.
- $d3$: Shipment of gold arrived in a truck.
- $q1$: gold silver truck.
- k-topics: = 2.

In this example, we use the $TFIDF$ as term weights and query weights. The following document indexing rules are also used in this example:

- Stop words, articles, punctuation marks, or numbers were removed.
- Text was tokenized and lower-cased.
- Text was split into terms.
- The stemming was performed (*e.g.* removing word endings) via WordNet [Fellbaum, 1998].
- Terms were sorted alphabetically.

❶ We use LSI to rank these documents (d1, d2 and d3) for the query "*gold silver truck*". As a

Table 2.13 : The term-document matrix and term-query matrix.

Terms	Documents		
	D1	D2	D3
a	1	1	1
arrived	0	1	1
damaged	1	0	0
delivery	0	1	0
fire	1	0	0
gold	1	0	1
in	1	1	1
of	1	1	1
shipment	1	0	1
silver	0	2	0
truck	0	1	1

Terms	Q1
a	0
arrived	0
damaged	0
delivery	0
fire	0
gold	1
in	0
of	0
shipment	0
silver	1
truck	1

first step, we set term weights and construct the term-document matrix TDM and term-query matrix TQM (*cf.* Table 2.13).

❷ As a second step, we decompose TDM matrix and find the T, S and D matrices, where $TDM = T_{m\times r} \cdot S_{r\times r} \cdot D_{r\times n}$ (*cf.* Table 2.14).

Table 2.14 : The T, S and D matrices.

-2.88051	-2.76080	-3.37152
-0.17259	0.03187	0.49087
-0.00521	0.64990	-0.21564
-0.43357	-0.01189	-0.08303
-0.00521	0.64990	-0.21564
-0.01449	0.27612	0.44193
0	0	0
0	0	0
-0.01449	0.27612	0.44193
-0.86715	-0.02379	-0.16607
-0.17259	0.03187	0.49087

0.31596	0	0
0	0.23872	0
0	0	0.10955

-0.01049	0.98855	-0.15052
-0.99758	-0.02067	-0.06624
-0.06859	0.14947	0.98638

❸ In a third step, we implement a rank 2 approximation ($k = 2$) by keeping the first two columns of T and D, and the first two columns and rows of S (*cf.* Table 2.15).

Table 2.15 : The T_k, S_k and D_k matrices (TSD').

-2.88051	-2.76080
-0.17259	0.03187
-0.00521	0.64990
-0.43357	-0.01189
-0.00521	0.64990
-0.01449	0.27612
0	0
0	0
-0.01449	0.27612
-0.86715	-0.02379
-0.17259	0.03187

0.31596	0
0	0.23872

-0.01049	0.98855
-0.99758	-0.02067
-0.06859	0.14947

❹ In the fourth step, we find the new document vector coordinates in this reduced 2-

[7] We should mention here, that all values in this example are taken from our prototype.

dimensional space. Rows of D_k hold eigenvector values. These are the coordinates of individual document vectors, hence:

- d1(-0.01049, 0.98855)
- d2(-0.99758, -0.02067)
- d3(-0.06859, 0.14947)

❺ In the fifth step, we find the new query vector coordinates in the reduced 2-dimensional space (*cf.* Table 2.16).

$$q = q^T T_k S_k^{-1} \tag{2.9}$$

Table 2.16 : The new query vector coordinates in the reduced 2-dimensional space.

$$(q^T)\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}\ (T_k)\ \begin{bmatrix} -2.88051 & -2.76080 \\ -0.17259 & 0.03187 \\ -0.00521 & 0.64990 \\ -0.43357 & -0.01189 \\ -0.00521 & 0.64990 \\ -0.01449 & 0.27612 \\ 0 & 0 \\ 0 & 0 \\ -0.01449 & 0.27612 \\ -0.86715 & -0.02379 \\ -0.17259 & 0.03187 \end{bmatrix}\ (S_k^{-1})\ \begin{bmatrix} \frac{1}{0.31596} & 0 \\ 0 & \frac{1}{0.23872} \end{bmatrix}\ (=)\ \begin{bmatrix} -1.08506 & 0.13788 \end{bmatrix}\ (q)$$

These are the new coordinates of the query vector in two dimensions. Note how this matrix is now different from the original term-query matrix TQM given in Step 1.

❻ In the last step, we rank documents (d1, d2 and d3) in decreasing order of query-document cosine similarities. In this example, we use 0.70 as a threshold for cosine similarity.

$$\cos\theta_{0.70}^{(q,d_j)} = \frac{\overrightarrow{d_q}\cdot\overrightarrow{d_j}}{|\overrightarrow{d_q}||\overrightarrow{d_j}|}$$

$$\textit{cosine similarity}(q,d_1) = \frac{(-1.08506)(-0.01049)+(0.13788)(0.98855)}{\sqrt{(-1.08506)^2+(0.13788)^2}\times\sqrt{(-0.01049)^2+(0.98855)^2}} = -0.0541$$

$$\textit{cosine similarity}(q,d_2) = \frac{(-1.08506)(-0.99758)+(0.13788)(-0.02067)}{\sqrt{(-1.08506)^2+(0.13788)^2}\times\sqrt{(-0.99758)^2+(-0.02067)^2}} = 0.98919 \approx 0.99$$

$$\textit{cosine similarity}(q,d_3) = \frac{(-1.08506)(-0.06859)+(0.13788)(0.14947)}{\sqrt{(-1.08506)^2+(0.13788)^2}\times\sqrt{(-0.06859)^2+(0.14947)^2}} = 0.4478$$

❖ We can see that document $d2$ scores higher than $d3$ and $d1$. Its vector is closer to the query vector than the other vectors. Thus, the descending order of documents based on the value of the similarity to the query, is as follows: $(d2 > d3 > d1)$.

2.4 Conclusion

In this chapter, we presented the context of our work, namely SPLE and software product variants. We also presented the background needed to understand the techniques used in our approach: FCA, RCA and LSI. We used these three techniques as a basis for our approach. We presented an example for each technique for better understanding.

CHAPTER 3

STATE OF THE ART

Competition is not only the basis of protection to the consumer,
but is the incentive to progress.
Herbert Clark HOOVER

Preamble

In this chapter, we present the state of the art relevant to our contributions. Section 3.1 gives an introduction of this chapter. Section 3.2 explains the main concepts related to our study. Section 3.3 presents the related work. Section 3.4 concludes this chapter by providing a concise overview of the different approaches and shows the need to propose REVPLINE approach.

3.1 Introduction

Research works on SPL and software variants have brought many advances in how to achieve complex software development by reusing the software artifacts such as feature, architectural elements, design entities and source code elements. This dissertation aims to reverse engineering FM from the source code of software variants. In order to achieve this goal we present three main contributions. The proposed approach consists of three main steps: ❶ feature mining (or feature location), ❷ documenting the mined feature implementations and ❸ reverse engineering FM from the mined and documented features. In this chapter, we present the state of the art relevant to our contributions. Firstly, we present the main concepts relevant to feature location, source code documentation and reverse engineering FM approaches then, we present the related work.

3.2 Key Concepts

In this section, we present the main concepts relevant to feature location, source code documentation and reverse engineering FM.

3.2.1 Feature Location

Definition 3.1. *Feature location is the activity of identifying an initial location in the source code that implements functionality in a software system [Dit* et al.*, 2013].*

According to Yoshimura et al. [Yoshimura *et al.*, 2006] the portion of functional commonality among two software products is about 60-75%; their implementations, however, share as little as around 30% of code. Feature location techniques aim at locating software artifacts that implement specific program functionality, *a.k.a.* a feature. In our work and based on our knowledge we must mention that feature location concept also can be known as feature mining or feature identification. Feature location techniques mainly employ textual, static, and dynamic analysis. Textual approaches such as [Poshyvanyk and Marcus, 2007] analyze words in source code using IR techniques. Static analysis approaches such as [Marcus *et al.*, 2004] examine structural information such as program convergence, control and data dependencies. Dynamic analysis approaches such as [Eisenberg and De Volder, 2005] examine execution traces of feature specific execution scenarios. Hybrid approaches such as [Zhao *et al.*, 2006] combine two or more types of analysis (*i.e.* textual, static and dynamic analysis) with the goal of using one type of analysis to compensate for the limitations of another, thus achieving better results.

Definition 3.2. *Traceability is the ability to describe and follow the life cycle of an artifact (e.g. requirements, design models, source code) created during the software life cycle in both forward and backward directions [Gotel and Finkelstein, 1994].*

Traceability link recovery pursues to connect different types of software artifacts (*i.e.* documentation with source code), while feature location is more worried with identifying source code associated with functionalities, not with specific sections of a document. In traceability link recovery contexts, the high-level descriptions of features are previously known and only the code that implements them is unknown [Dit *et al.*, 2013]. Our studies are limited to code-to-document(*e.g.* feature, use-case) traceability link.

Feature location supports developers during various activities such as software maintenance. No maintenance task can be completed without first locating and understanding the code that is relevant to the task at hand [Dit *et al.*, 2013]. Software maintenance involves adding new features to a system, improving and reengineering existing features and removing unwanted features (*e.g.* bugs) [Xue, 2013]. Software maintenance usually takes 70% of overall project costs [Xue *et al.*, 2012].

3.2.2 Source Code Documentation

Several concepts are closely related to the source code documentation, such as program understanding and source code comprehension. We present here these main concepts.

Definition 3.3. *Software comprehension is the process whereby a software practitioner understands a software artefact using both knowledge of the domain and/or semantic and syntactic knowledge, to build a mental model of its relation to the situation [Müller* et al.*, 1993].*

Comprehending software is one of the core software engineering activities. Software comprehension is required when a programmer maintains, reuses, migrates or enhances software systems. Software comprehension (also known as "program understanding" or "source code comprehension") is the process of taking computer source code and understanding it. According to Rajlich and Wilde [Rajlich and Wilde, 2002] software that is not comprehended cannot be changed.

Source code documentation is very important in our work. The documentation process aims to assign for each feature implementation its name and description. In order to reverse engineering FM from the source code of software variants, and after feature mining process, it is very important to document the mined feature implementations. In our work, we rely on the use-case diagram of software variants to document the mined feature implementations. In case of absence of the use-case diagram, we rely on the source code elements (OBE or identifier names) to document each feature.

The comprehension or documentation of feature implementation is a complex problem-solving task [Müller *et al.*, 1993]. In order to give a precise definition of the feature documentation, we consider that the feature documentation is the process of taking feature implementation and understanding it by providing name or a more detailed description based on software artifacts such as: use-case diagram or identifier names.

Existing approaches extract labels, topics, names, software structure, traceability link or code summarization from the software in order to facilitate the comprehension of the code. Software comprehension plays an important role in maintenance tasks [Rajlich and Wilde, 2002]. For purposes of constructing a FM and reusing existing features in other software, each feature implementation that is presented to the human user must have a meaningful name. In addition, feature documentation is needed in order to understand existing software variants and facilitate their maintenance.

3.2.3 Reverse Engineering Feature Model

There are many concepts that are relevant to the reverse engineering, such as forward engineering and re-engineering. We present here the main concepts related to reverse engineering and some concepts that are relevant to our work.

Definition 3.4. *Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction [Chikofsky and Cross II, 1990].*

Reverse engineering involves analyzing a subject system in order to determine its components and the relations between those components. It also involves the creation of alternative representations of the system, usually at a higher level of abstraction. Reverse engineering does not involve changing or replicating the system; it is only concerned with an examination of the system and can occur at any stage in the software development life cycle.

Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system [Chikofsky and Cross II, 1990]. Forward engineering is the process of moving through the stages of design, starting at the highest level of abstraction moving to a specific implementation. In the context of SPL, the software derives from the core assets based on the selected features in the feature model.

Re-engineering is the process of examination and alteration of a system to reconstitute it in a new form. Re-engineering involves both forward engineering and reverse engineering [Chikofsky and Cross II, 1990]. The concept of reengineering is well-suited to the process of obtaining an SPL from software variants. There are other related terms such as migration. Migration is a term used more in the context of changing languages, databases, or platforms, often using automated converters as a way of transforming software variants [Laguna and Crespo, 2013]. Software re-engineering is concept with a long history. The primary goal was to reach new levels of efficiency of the existing assets, without recurring to the development of new systems from scratch.

Existing approaches extract FMs from two levels: low level model (*i.e.* source code) and high level models (*e.g.* product description, product configurations, requirements, *etc.*). In this dissertation our goal is to reverse engineering a feature model from the source code of software product variants.

3.3 Related work

In this section, we present the related work. We provide a concise overview of the different approaches and we show the need to propose our REVPLINE approach. Based on the research contributions, we have identified three kinds of related work: feature location approaches, source code documentation approaches and reverse engineering FM approaches.

3.3.1 Feature Location Approaches

In this section, we present the closest approaches that are related to the feature location in the object-oriented source code. This section presents feature location approaches in software family (*cf.* Section 3.3.1.1) then in a single software system (*cf.* Section 3.3.1.2). Finally, we present a synthesis and comparison table between these approaches (*cf.* Section 3.3.1.3). In this section we present feature location in software family before single software; the reason behind this choice is that our approach relevant to software variants not to single software. We present the selected papers in Table 3.1 clarified by their target software (family versus single).

We evaluate the studied works according to the following *criteria*: objectives of the study (feature location or traceability link), target software (single software versus software family),

Table 3.1 : Summary of the feature location approaches (the selected papers).

ID	Reference	Feature Location	
		Software Family	Single Software
1	[Ziadi *et al.*, 2012]	✗	
2	[Rubin and Chechik, 2012]	✗	
3	[Duszynski *et al.*, 2011]	✗	
4	[Xue, 2011]	✗	
5	[Xue *et al.*, 2012]	✗	
6	[Linsbauer *et al.*, 2013]	✗	
7	[Salman *et al.*, 2013]	✗	
8	[Marcus *et al.*, 2004]		✗
9	[Poshyvanyk and Marcus, 2007]		✗
10	[Eisenberg and De Volder, 2005]		✗
11	[Zhao *et al.*, 2006]		✗
12	[Paškevičius *et al.*, 2012]		✗
13	[Rubin and Chechik, 2013b]		✗
14	[Dit *et al.*, 2013]		✗

programmed method (automatic versus semi-automatic), type of code analysis (static, dynamic, textual or hybrid), inputs (source code, feature description), techniques (LSI, FCA, *etc.*), outputs (feature implementation, code-to-feature traceability link, *etc.*), target languages (Java, C or C++), type of evaluation (case study or empirical data), reduce search space and tool support.

3.3.1.1 Feature Location Approaches in Software Family

We present here a collection of relevant approaches to feature location in a set of software product variants. In this section, we present each approach separately, and we do a synthesis in section 3.3.1.3.

■ Ziadi *et al.* [Ziadi *et al.*, 2012] proposed a semi-atomic approach to identify features from object-oriented source code. Their approach takes as input the source code of a set of product variants but instead of analyzing the source code itself, they rely on higher-level abstractions. A structural model (*i.e.* a simplified UML class diagram) of each product is first extracted by reverse engineering. This model is then decomposed into a set of atomic pieces where each atomic piece is an elementary model construction primitive (CP).

The construction primitives that they use to decompose the UML class diagram concern the main elementary elements in class diagrams. This includes the set of construction primitives: package, class, attribute, method. They assumed in their work that the product variants use the same vocabulary to name packages, classes, attributes and methods in its source code.

They propose an ad hoc algorithm to identify the feature candidates of a given set of products. Their algorithm gathers all construction primitives that are common to all product variants as one feature (*i.e.* base feature). For the construction primitives that are unique to a single product or common to two or more products but not all products, their algorithm gathers them as one feature.

Their approach only investigates products in which the variability is represented in the name of packages, classes, attributes and methods. Their approach does not consider product variants in which the variability was mainly represented in the body of methods (different local variables, method invocations and attribute accesses). They neither consider the parameters of operations nor the types of attributes.

Their approach does not split mandatory features they are stored in one unique mandatory feature. They group optional features that are always together in the same set of product variants as one feature. They do not expect this problem to be a major one for several reasons. First, considering more products may be enough to solve it. Second, splitting some features is certainly much less time consuming and error-prone than browsing a large source code so that the impact of the problem is probably limited. They manually named the extracted feature implementations. They consider that feature implementations may overlap and they called the shared construction primitives between two or more features a *junction*. In our work, we will reuse this terminology.

■ Rubin *et al.* [Rubin and Chechik, 2012] proposed an approach focused on locating distinguishing features of software product families realized via code cloning. In reality software families often emerge ad-hoc, when companies have to release a new product that is similar to existing ones. In many cases, new products are created using code cloning mechanisms (the "clone-and-own" approach) when an existing product is copied and later modified independently from the original version. In cloning process, variants are created by duplicating a specific version and continuing its development independently from the original [Rubin *et al.*, 2012].

Their approach locate distinguishing features – those that are present in one but not all variants of a product family realized via code cloning. Their approach is limited to two software variants. The features of interest are implemented in the unshared parts of the program. Their work considers optional features only. They define a notion of a diff set – an artifact capturing unshared parts of the program of interest.

They focus on distinguishing features of a program P (*i.e.* those that exist in P but not in another variant $\overline{P}$). They know a priori that all code of such features is present in P but absent in $\overline{P}$. That is, the implementation of such features fully resides in diff sets. So the output of their approach is limited to common and unshared parts of source code obtained from comparing two program variants. Their approach does not require program execution. They rely on static analysis. Their approach does not distinguish between optional features that appear in unshared part of the source code. They validate their approach on a small but realistic example and describe initial evaluation results.

■ Duszynski *et al.* [Duszynski *et al.*, 2011] said that not every SPLs starts from the scratch. In reality, often companies face the problem that after a while their software is deployed in several variants and the need arises to migrate to systematic variability and variant management using a SPL approach. They describe a framework for the analysis and visualization of similarities across related systems. After identifying corresponding files, the framework facilitates browsing variants in large code bases.

They introduce variant analysis technique which visualizes commonalities and variabilities in the source code of multiple software product variants. The goal of the technique is to support reuse potential assessment by delivering precise quantitative information about the similarity across the analyzed software variants (*e.g.* number of classes). Their approach used to identify system parts suitable for transformation into reusable assets. In their work they define a visualization concepts that enable easy interpretation of the results at any level of abstraction (source code, files, subsystems, whole systems), even for many variants.

On the lowest level of detail, the variant analysis technique uses occurrence matrices for organizing variability information. In the occurrence matrices each variant is represented as a set of distinct atomic elements (classes). Then a matrix is created for each variant, the rows of the matrix represent the atomic elements of the variant, and the columns represent all the analyzed variants. Then union matrix is created for the union of all sets. Its rows represent all the elements existing in any of the sets. Each matrix cell has a value of "1" if the element represented by the field's row belongs to the variant represented by the field's column, or a value of "0" if not. Each matrix has an additional summary column, which counts the number of variants the given element belongs to. They define a method for finding and visualizing the relationship between code variants. The authors map elements between system variants using data models, element occurrence rates, and system models. The results are presented via Venn diagrams and bar graphs to show the overlap and variability in the systems.

Occurrence matrices are well-suited for analyses of variability distribution amongst software variants. The summary column, storing the number of variants the given element belongs to, gives direct information about the reusability of the element: if the value is equal to the number of all variants N, the element is a core element that is identical in all software variants. The elements with a value of 1 exist in only one variant and are unique to this variant. The elements with values in the $2...N-1$ range are shared across some, but not all of the software variants. For each element, it is known to which software variants it belongs and the corresponding elements from other variants can be easily located. The core, shared, or unique status of the element can be visualized in the respective compilation unit visualization.

■ Xue [Xue, 2011] addresses the problem of re-engineering of the legacy software variants into SPL. To migrate a family of legacy software products into SPL for effective reuse, one has to understand commonality and variability among existing product variants. Xue proposes an approach that consolidates feature knowledge from top-down domain analysis with bottom-up analysis of code similarities in subject software products. They proposed a method which integrates model differencing, clone detection and LSI (information retrieval technique), which can provide a systematic means to re-engineer the legacy software products into SPL based on automatic variability analysis.

Xue presents a model differencing based on methods to detect changes that occurred to product features in a family of product variants. The primary input to his approach is a set of product FMs. A product FM captures all the features and their dependencies in a product variant. Through this method he knows that a specific feature is unique for a specific product. In his work, the developers should document the features and their dependencies for each variant, which provides the input for the variability analysis at the requirement level. Then he uses clone differentiating. Clone differencing technique reports the differences between two clones code. The variability recovered from product FMs needs to be correlated with the variability identified from the clones. The difference in terms of feature at the requirement level should be connected to the difference in terms of clone at the implementation level. The underlying intuition of their approach is that the presence or absence of a feature in a product variant should be reflected in the presence or absence of certain design elements and code fragments.

Xue uses model differencing algorithm to identify evolutionary changes that occurred to features of different product variants at requirement and implementation levels, using a case study and empirical data. Xue approach needs as inputs FMs (*i.e.* features) and source code of software product variants. His approach is limited to variability (*i.e.* without commonality) of software variants at requirement and implementation levels.

■ Xue *et al.* [Xue *et al.*, 2012] extend the work of Xue [Xue, 2011]. They present an approach to support effective feature location in a collection of software product variants. The novelty of their approach is that they exploit commonalities and differences of product variants by software differencing and FCA techniques so that LSI (IR technique) can achieve satisfactory results for feature location in product variants. They have implemented their approach and conducted evaluation with a collection of nine Linux kernel product variants. Their evaluation shows that their approach always significantly outperforms a direct application of IR technique in the subject product variants.

For a product variant, their approach takes as input a set of features (*resp.* feature description) that the product variant supports and a static program model built from the implementation of the product variant. They assume that each *feature* of a product variant is identified by a name and is described using some natural language description. The static program model of software implementation is a graph. The node set contains code units of interest for feature location such as methods and data structures. Each code unit is associated with a set of properties such as identifiers and comments. The edge set contains relationships between code units, such as a function call and data-structure usage.

They identify the distinct features (*resp.* code units) in software product family, then they group features (*resp.* code units) into disjoint, minimal partitions by FCA (*i.e.* reduce search space), then they apply LSI to find the traceability link between feature description and its implementation (*i.e.* code-to-feature traceability link) in an efficient way. They exploit commonality and variability (*i.e.* at features and code units level) across software variants to reduce the search space and then apply the LSI technique. We should mention here that their approach represents the best one of the approaches that we studied (*i.e.* code-to-feature traceability link) based on the methods, techniques used and the results. Their approach is limited to variability of software variants (*i.e.* optional features). They assume that commonalities and differences between software variants can be determined statically. Their approach does not make any assumptions regarding how product variants are generated and managed (*resp.* features naming). It requires as input only a set of features and program models of software variants.

■ Linsbauer *et al.* [Linsbauer *et al.*, 2013] said that companies typically only know which features are implemented in which software variants but they do lack precise knowledge about the portions of code these features are implemented. They present a novel technique for deriving the traceability between features and code in software variants by matching code overlaps and feature overlaps. The technique they propose expects as input a number of software variants with their corresponding feature sets and code. They implemented their approach on the code granularity level of class methods and attributes.

The precision of the technique depends on the ability to distinguish individual features. They do not require a software variant for every possible feature combination. Rather, to distinguish any two features, they typically only require one product with and one without the feature. By matching these sets, they obtain code fragments that belong to individual features or groups thereof. In cases that a set of features always occur together their approach is unable to exactly distinguish what code fragment belongs to which feature individually. In their approach they assume that software variants which have common features also have common code, and that this common code implements exactly these common features (*i.e.* products that have features in common will also have code in common and vice versa). The automated traceability extraction is thus limited to code pieces with a unique trace.

They define an association as a tuple where the first element is the set of modules (features) and the second element is the set of code supported by product variant (association = (module-set, code-set). As an example, let us assume we have two product variants (product A and product B as input). Product A supports "line" and "wipe" features while product B support only "line" feature. The association A =({line, wipe}, {c1, c2, c3, c4, c5, c7, c8}) and association B = ({line}, {c1, c2, c3, c4, c5, c7}). Associations are then intersected by respectively intersecting their module sets and their code sets. Product B consists of only one module, which is the base module line. The code these two products have in common is therefore associated with the base module line (line ↔ c1, c2, c3, c4, c5, c7). The remaining code in product A is left for the base module wipe (wipe ↔ c8). They evaluate their approach on three case studies of different sizes and complexity. More than 99% of the code pieces were correctly assigned in a matter of seconds.

■ Eyal-Salman *et al.* [Salman *et al.*, 2013] propose an approach to identify code-to-feature traceability links to a collection of software product variants. The approach expects as input a number of software variants with their corresponding source code and feature sets (*resp.* feature description). They implemented their approach on the code granularity level of class only. They apply their approach on two case studies.

In their work traceability link recovery process consists of four main steps. The first step aims to reduce LSI spaces. They split classes (*resp.* features) into common and variable partitions. For the variable partition of classes they rely on FCA to split it into minimal disjoint sets. In the second step code-topics are derived from each minimal disjoint set of classes that are computed in the previous step. In this step, they follow two steps to derive the code-topics: computing similarity among classes and grouping similar classes into code-topics using FCA. The code-topic is a cluster of similar classes that are grouped together to cover the same topic. In the third step, the traceability links between features and their possible corresponding code-topics are established using LSI. Finally, by determining code-topics related to each feature, they determine all classes that implement each feature by decomposing each code-topic to its classes.

In their work they use two kinds of source code information to compute the similarity between classes: textual and structural information. Textual similarity among given classes refers to lexical matching between terms derived from identifiers related to these classes. They depend on Vector Space Method (VSM) to compute the textual similarity. For structural similarity, they consider two classes are structurally similar if they have at least one of the following relations: inheritance, method call, shared method invocation and shared attribute access relations. Their work is based on methods and techniques that are very close to Xue *et al.* [Xue *et al.*, 2012] except the use of lexical and structural similarity to extract the code-topics from each minimal disjoint set. In addition their work doesn't deal with commonality. There are many missing links in the obtained results. For example, *logging* feature in ArgoUML is missing. The reason behind these missing features is that the implementation of these features is at method body level not at the class level.

3.3.1.2 Feature Location Approaches in Single Software

We present here a collection of relevant approaches to the feature location in a single software system. We present each approach separately, and we do a synthesis in section 3.3.1.3. Finally, we present two surveys relevant to feature location in a single software system.

❖ Marcus *et al.* [Marcus *et al.*, 2004] introduce one of the first approaches for using IR techniques for feature location. The approach is based on using domain knowledge embedded in the source code through identifier names (*e.g.* method and attribute) and internal comments. The analyzed program is represented as a set of text documents describing software elements (*e.g.* methods or data type declarations). To create this set of documents (*i.e.* corpus), the system extracts identifiers from the source code and comments, and separates the identifiers using known code styles (*e.g.* the use of underline _ to separate words). Each software element is described by a separate document containing the extracted identifiers and translated to LSI space vectors using identifiers as terms.

The technique requires no user interaction besides the definition and the refinement of the input query. The proposed approach is based on the textual analysis of the source code. Given a natural language query containing one or more words, identifiers from the source code, a phrase or even short paragraphs formulated by the user to identify a feature of interest, the system converts it into a document in LSI space, and uses the similarity measure between the query and documents of the corpus in order to identify the documents most relevant to the query.

In order to determine how many documents the user should inspect, the approach partitions the search space based on the similarity measure: each partition at step $i+1$ is made up of documents that are closer than a given threshold α (based on their experience, $\alpha = 0.075$ gives good results) to the most relevant document found by the user in the previous step i. The user inspects the suggested partition and decides which documents are part of the concept. The algorithm terminates once the user finds no additional relevant documents in the currently inspected partition and outputs a set of documents that were found relevant by the user, ranked by the similarity measure to the input query.

❖ Poshyvanyk *et al.* [Poshyvanyk and Marcus, 2007] extend the work of Markus *et al.* [Marcus *et al.*, 2004] with FCA to select most relevant, descriptive terms from the ranked list of documents describing source code elements. That is, after the documents are ranked based on their similarity to the input query using LSI, as in [Marcus *et al.*, 2004], the system selects the first n documents and ranks all terms that appear uniquely in these documents. The ranking is based on the similarity between each term and the document of the corpus, such that the terms that are similar to those in the selected n documents but not to the rest are ranked higher. Terms that are similar to documents not in the selected n results are penalized because they might be identifiers for data structures or utility classes which would pollute the top ranked list of terms. After the unique terms are ranked, the system selects the top k terms (attributes) from the first n documents (objects) and applies FCA to build the set of concepts. The user can inspect the generated concepts – the description and links to actual documents in the source code – and select those that are relevant. Similarly to [Marcus *et al.*, 2004], the technique requires a low amount of user interaction.

❖ Eisenberg *et al.* [Eisenberg and De Volder, 2005] present an effort to deal with the difficulty of scenario definition. The approach accepts that the user is unfamiliar with the system and thus should use pre-existing test suites, such as those usually available for systems developed with a Test Driven Development (TDD) strategy. The approach accepts as input a test suite that has some correlation between features and test cases. In their work all features are exercised by at least one test case. Tests that exhibit some part of feature functionality are mapped to that feature and denoted to as its exhibiting test set. Tests which are not part of any exhibiting test set are grouped into sets based on similarity between them and are referred to as the non-exhibiting test set.

For each feature, the system gathers execution traces obtained by running all tests of the feature's exhibiting test set and generates a calls set which lists (caller/callee) pairs for each method call specified in the collected traces. It then ranks each method heuristically. For each feature, both the ranked list of methods and the generated call set are returned to the user. The goal of the former is to rank methods by their relevance to a feature, whereas the goal of the latter is to assist the user in understanding why a method is relevant to a feature. This approach based on execution scenarios. The underlying technology is trace analysis. The program represents as executable form. The input of this approach is set of test cases. This approach requires user interaction.

❖ Zhao *et al.* [Zhao *et al.*, 2006] accept a set of feature descriptions as input and focuses on locating the specific and the relevant functions of each feature using program dependence analysis (PDA) and VSM (IR technique). The specific functions of a feature are those definitely used to implement it but are not used by other features. The relevant functions of a feature are those involved in the implementation of the feature. This approach requires user interaction. This approach considers as hybrid approaches because it combines two types of analysis (textual and dynamic analysis (execution traces)).

The analyzed program is represented as a Branch-Reserving Call Graph (BRCG) – a development of the call graph with branching and sequential information, which is used to construct the pseudo execution traces for each feature. Each node in this graph is a function, a return statement, or branch. Loops are viewed as two branch statements: one going through the loop body and the other one exiting directly. The nodes are related either conditionally, for alternative outcomes of a branch, or sequentially, for statements executed one after another. Their approach receives a paragraph of text as a description of each feature. The text can be obtained from the requirements documentation or from domain experts. It transforms each feature description into a set of index terms (considering only nouns and verbs). These will be used as documents. The system then mines the names of each method and its parameters, separating identifiers using known coding styles and transforms them into index terms. These will be used as queries (*i.e.* feature).

To reveal the connections between features and functions, documents (*i.e.* feature descriptions) are ranked for each query (function) using the VSM. The similarity between a document and a query is computed as a cosine of the angle between their corresponding vectors. For each document (*i.e.* feature description), the system creates a sorted list of queries (*i.e.* functions), ranked by their similarity degrees and identifies a pair of functions with the largest difference between scores. All functions before this pair, called a division point, are considered initial specific functions to the feature. Then, the approach analyzes the program's BRCG and filters out all branches that do not contain any of the initial specific functions of the feature, because those are likely not relevant; all remaining functions are marked as relevant. Functions relevant to exactly one feature are marked as specific to that feature.

Surveys of feature location techniques: ❶ Rubin and Chechik [Rubin and Chechik, 2013b] provide a detailed description of twenty-four feature location techniques in single software and discussed their properties. All of the surveyed approaches share the same goal – establishing traceability between a specific feature of interest that is specified by the user and the artifacts that implement that feature, their underlying design principles, their input, and the quality of the results which they produce differ substantially. They also illustrated the techniques on a common example in order to improve the understandability of their underlying principles and implementation decisions. The goal of their survey is to show that none of the existing feature location techniques in the survey are designed to consider families of related products and only treat different products of a product line as individual, unrelated entities. Thus they discuss possible directions for leveraging SPLE architectures in order to improve the feature location process. ❷ Dit *et al.* [Dit *et al.*, 2013] present an inclusive survey of feature location techniques in single software. In this survey eighty nine articles from 25 venues have been reviewed and classified within the taxonomy in order to organize and structure existing work in the field of feature location. The survey also discusses open issues and defines future directions in the field of feature location. This survey classifies existing approaches using some criteria (*i.e.* code analysis, input, technique, output, programming language and the evaluation way). We use the same criteria to compare feature location studies and other criteria.

3.3.1.3 Synthesis

Here, we summarize the related work for feature location according to the criteria presented at the beginning of this section. Table 3.2 presents a comparison between feature location studies. In the following, we mention high level remarks on the related work from Table 3.2.

- ❏ Feature location in a collection of product variants typically relies on reduce the search space by exploiting commonality and variability across software variants at the source code and feature levels.
- ❏ The majority of existing approaches are designed to locate the program elements of a particular feature in a single software system.
- ❏ Some existing approach accepts as input two sources of information such as source code of software variants and feature/feature description.
- ❏ Some existing approaches investigate variability at package or class levels. These approaches are applied at different level of granularity (*e.g.* class, attribute, *etc.*).
- ❏ FCA and LSI are the most used techniques in feature location. Static analysis of source code is the most used type of code analysis.
- ❏ The development paradigm of all studies is object-oriented. Most studies are programmed in a way semi-automatic and need user interaction.
- ❏ The majority of approaches are evaluated through the use of famous evaluation metrics such as precision, recall or F-measure.

Table 3.2 : Summary of feature location studies (comparison table).

ID	Reference	Objectives		Software		Programmed method		Code analysis					Inputs									Techniques										Outputs								Languages			Evaluation		Reduce search space	Tool support
		Feature location	Traceability link	Single software	Software family	Automatic	Semi-automatic	Static	Textual	Dynamic	Hybrid	UML class diagram	Package	Class	Attribute	Method	Method body	Feature name	Feature description	Query	Scenario	FCA	LSI	VSM	Clone detection	Code dependencies	Ad hoc algorithm	Program dependence analysis	Program differencing	Variant analysis	Trace analysis	Junction	Common code	Variable code	Mandatory feature	Optional feature	Traceability link	Visualization	Ranked methods	Java	C	C++	Case study	Empirical data		
1	[Ziadi *et al.*, 2012]	✗			✗		✗					✗	✗	✗	✗	✗											✗					✗			✗	✗				✗			✗			
2	[Rubin and Chechik, 2012]	✗			✗	✗		✗						✗											✗								✗	✗										✗		
3	[Duszynski *et al.*, 2011]	✗			✗	✗		✗						✗																✗			✗	✗				✗				✗	✗			
4	[Xue *et al.*, 2012]		✗		✗	✗		✗								✗	✗	✗	✗			✗	✗						✗								✗				✗		✗		✗	✗
5	[Linsbauer *et al.*, 2013]		✗		✗		✗	✗						✗	✗	✗		✗							✗		✗										✗						✗			
6	[Salman *et al.*, 2013]		✗		✗		✗	✗						✗				✗	✗			✗	✗	✗		✗											✗			✗			✗		✗	✗
7	[Marcus *et al.*, 2004]	✗		✗			✗	✗								✗	✗			✗			✗																✗	✗	✗	✗	✗			
8	[Poshyvanyk and Marcus, 2007]	✗		✗			✗		✗											✗		✗	✗															✗	✗	✗			✗			✗
9	[Eisenberg and De Volder, 2005]	✗		✗			✗			✗											✗										✗							✗	✗	✗			✗			
10	[Zhao *et al.*, 2006]	✗		✗			✗				✗					✗				✗			✗	✗				✗											✗		✗		✗			

3.3.2 Source Code Documentation Approaches

In this section, we present source code documentation approaches in single software (*cf.* Section 3.3.2.1), then we present code-to-document traceability link approaches in single software (*cf.* Section 3.3.2.2) and feature documentation approaches in software variants (*cf.* Section 3.3.2.3). Finally in Section 3.3.2.4 we present a synthesis and comparison table between these approaches. We present the selected papers in Table 3.3.

Table 3.3 : Summary of source code comprehension studies (the selected papers).

ID	Reference	Source code comprehension	
		Single Software	Software Family
1	[Kebir *et al.*, 2012]	✗	
2	[Kuhn, 2009]	✗	
3	[Kuhn *et al.*, 2007]	✗	
4	[Lucia *et al.*, 2012]	✗	
5	[Haiduc *et al.*, 2010]	✗	
6	[Falleri *et al.*, 2010]	✗	
7	[Grechanik *et al.*, 2007]	✗	
8	[Marcus and Maletic, 2003]	✗	
9	[Diaz *et al.*, 2013]	✗	
10	[Sridhara *et al.*, 2010]	✗	
11	[Davril *et al.*, 2013]		✗
12	[Yang *et al.*, 2009]		✗
13	[Paškevičius *et al.*, 2012]		✗
14	[Ziadi *et al.*, 2012]		✗

We evaluate the studied works according to the following *criteria*: objectives of the study (code comprehension, *etc.*), target software (single software versus software family), programmed method (automatic versus semi-automatic), type of code analysis (*e.g.* static, dynamic), inputs (source code, use-case diagrams, *etc.*), techniques (LSI, VSM, *etc.*), outputs (topics, labels, terms, *etc.*), target languages (*e.g.* Java), type of evaluation (case study or empirical data) and tool support.

3.3.2.1 Code Documentation Approaches

We present here a collection of relevant approaches to the source code documentation in single software. We present each approach separately. We present the approaches that extract *labels*, *topics*, *names*, *software structure*, *code summarization* or *traceability link* from software system code in order to facilitate the comprehension of the code. Then we propose a synthesis of these approaches in Section 3.3.2.4.

■ Kebir *et al.* [Kebir *et al.*, 2012] propose an approach to identify components from object-oriented source code of single software. Their approach proposed allocating names to the components based on the class names. Their work identifies component names in three steps: extracting and tokenizing class names from the identified cluster, weighting words and constructing the component name by using the strongest weighted tokens.

In the first step, class names are split into words according to the camel-case syntax. For example: StringBuffer is split into String and Buffer. In the second step, a weight is assigned to each extracted token. A large weight is given to tokens that are the first word of a class name. A medium weight is given to tokens that are the first word of an interface name. Finally a small weight is given to the other tokens. In the third step, a component name is constructed using the strongest weighted tokens. The strongest weighted token is the first word of the component name; the second strongest weighted word is 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.

Their heuristic is based on the following observation: in many object-oriented languages, class names are a sequence of nouns concatenated using a camel-case notation (*e.g.* Network-Settings). The first word of a class name indicates the main purpose of the class; the other words indicate a complementary purpose of the class and so on. To summarize the work, their approach accepts source code of single software. Then they identify its components and at last, they document the identified components and interfaces using class names.

■ Kuhn [Kuhn, 2009] presents a lexical approach that uses the log-likelihood ratio of word frequencies to automatically retrieve labels from source code. The approach can be applied to compare components with each other (*i.e.* extract labels to describe their differences as well as commonalities), to compare a component against a normative corpus (*i.e.* providing labels for components), and to compare different revisions of the same component (*i.e.* documenting the history of a component).

Their approach is applicable at any level of granularity, from the level of projects down to the level of methods. They extract the names of packages, classes (including interfaces), fields, methods and type parameters. Then they split the extracted names by camel-case to accommodate to the Java naming convention. They use word frequencies in code to automatically label software components. By using log-likelihood ratio of word frequencies they extract labels from source code and use them to label software components.

The idea behind log-likelihood ratio is to compare two statistical hypotheses, of which one is a subspace of the other. Given two text corpora, they compare the hypothesis that both corpora have the same distribution of term frequencies with the hypothesis given by the actual term frequencies. They use log-likelihood ratio to compare between two text corpora to distinguish between terms specific to the first corpus and terms specific to the second corpus (*resp.* common terms).

■ Kuhn *et al.* [Kuhn *et al.*, 2007] propose the use of information retrieval to exploit linguistic information found in source code, such as identifier names and comments. In order to enrich software analysis with the developer knowledge that is hidden in the code naming they proposed this approach. They introduce semantic clustering, a technique based on LSI and clustering to group source artifacts (*i.e.* classes) that use similar vocabulary. They call these groups semantic clusters and they interpret them as linguistic topics that reveal the intention of the code. They compare the topics to each other, identify links between them, provide automatically retrieved labels (they employ LSI again to automatically label the clusters with their most relevant terms), and use a visualization to illustrate how they are distributed over the system. They use distribution maps to illustrate how the semantic clusters are distributed over the system. Their work is language independent as it works at the level of identifier names.

Semantic clustering is a method proposed to group similar classes of a system, according to their vocabularies. In their work, every class is represented as a document, and terms are obtained by extracting and filtering identifiers and comments. Furthermore, terms are later weighted with TF-IDF function (The most common weighting scheme), in order to punish words that appear in many documents of the vocabulary. The resulting term-document matrix is then processed by LSI (an IR technique). In the term-document matrix, each class is retrieved as a vector, and the similarity of a pair of classes is calculated by the cosine of the smallest angle formed by their representing vectors. Then, the technique relies on a hierarchical clustering to group a fixed number of clusters which has similarity greater than a given threshold. Lastly, queries are used to retrieve a set of words representing the meaning of each group (called semantic cluster) and the distribution map is generated.

To summarize the work, authors present the concept of semantic clustering, a technique based on LSI to group source code documents that share a similar vocabulary. By applying LSI to the source code, the documents are clustered based on their similarity into semantic clusters, resulting in clusters of documents that implement similar functionalities. The authors also used LSI to label the identified clusters. Finally, a visual notation is provided aiming at giving an overview of all the clusters and their semantic relationships. If the developers did not name the identifiers with care (*e.g.* a, b and c), their approach fails, since the developer knowledge is missing. The extracted topics are used to describe functionality of a given segment of code.

■ De Lucia *et al.* [Lucia *et al.*, 2012] propose an approach for source code labelling, based on IR techniques, to identify relevant words in the source code of single software. They applied various IR methods (such as VSM, LSI and Latent Dirichlet Allocation (LDA)) to extract terms from class names by means of some representative words, with the aim of facilitating their comprehension or simply to improve visualization. This work investigates to what extent an IR-based source code labeling would identify relevant words in the source code, compared to the words a human would manually select during a program comprehension task.

The study was organized in three steps. In the first step, they asked developers to describe the selected source code classes with a set of 10 keywords. Then, they applied different techniques (IR techniques) to automatically extract keywords from the selected classes. Once they have the set of keywords identified by the developers and the set of keywords identified by the experimented technique, they compare them in order to compute the overlap and try to answer this question (goal of the study): To what extent terms selected by developers to describe a source code artifact overlap with those that can be identified by an automated technique? Results show that, overall, automatic labeling techniques are able to well-characterize a source code class, as they exhibit a relatively good overlap (ranging between 50% and 90%) with the manually-generated labels.

The achieved results propose that the different elements of a class (*e.g.* class name, method signatures, local variables, and comments) play different roles during the automatic labeling of source code. Basically, the most important words are included in the class name, in the method signature, and in the attribute names (*i.e.* the high-level structure of the class). Comments play an important role, but only when they have a good verbosity.

■ Haiduc *et al.* [Haiduc *et al.*, 2010] proposed a technique for automatically summarizing source code, leveraging the lexical and structural information in the code. The goal of their approach is the automatic generation of summaries for source code entities. Summaries are obtained from the contents of a document by selecting the most important information in that document. The approach is based on using lexical and structural information from the source code.

Lexical information (*i.e.* identifiers and comments) is extracted from the source code entity to be summarized (*i.e.* package, class, method, function, *etc.*). Common English and programming language keywords are removed, identifiers are split in their constituent words, and stemming is used to replace English words with their root. The source code is transformed into a text corpus, where each document corresponds to a source code entity. They use text retrieval (TR) technique (LSI) to determine the most important *n* terms for each document.

Their approach is not just an instance of text summarization, just as source code is not just text. One of the requirements for the source code summaries is to reflect the domain semantics as well as the programming language semantics. They attach structural information (*e.g.* method m1 calls method m2, class c declares method m, class c1 extends class c2, class c implements interface I, *etc.*) to the words selected by the TR method, specifying what role each term plays in the source code, such as class name, method name, parameter name, parameter type, local variable, etc. This type of information is used not only to enrich the summary but also to help build it. The top *n* words selected with the TR method together with the structural information form the summary.

For the automatic summarization, they used only lexical information in the proposed approach. They used LSI as the TR technique, as it is one of the most commonly used techniques in automatic natural language summarization. The first step in determining the automatic summaries was to use LSI for indexing the corpus obtained by considering each method in the source code as a separate document. Then, the cosine distance between the text of the method and each of the terms in the corpus was computed in the LSI-reduced space. The corpus terms were then ordered in decreasing order based on their similarity with the method, so that the term with the highest cosine similarity to the method would be the first in the list. After this, they constructed the summary by considering the top five terms in the ordered list.

■ Falleri *et al.* [Falleri *et al.*, 2010] proposed a wordNet-like approach to extract the structure of single software using the relationships among identifier names. The approach considers natural language processing techniques which consist of tokenization process (straightforward decomposition technique by word markers, *e.g.* case changes, underscore, *etc.*), part of speech tagging, and rearranges order of terms by the dominance order of term rules based on part of speech.

Their work is based on natural language processing techniques that automatically extract and organize concepts from software identifiers in a WordNet-like structure: lexical views. Those lexical views give useful insight on an overall software architecture and can be used to improve results of many software engineering tasks. Their approach can be applied at any level of granularity (*e.g.* class identifiers, attribute identifiers, *etc.*). The only thing required by their approach is a set of identifiers.

Their approach automatically classifies a set of identifiers in a WordNet-like structure (*i.e.* fully automated approach). They call this structure a lexical view. Their approach is performed as the following: cut up identifiers in order to find the primitive words they are composed of, then, classify the previously extracted primitive words into lexical categories (*e.g.* noun, verb, adjective), then, apply rules specific to the English language to determine which words are dominant and impose the meaning of the identifiers, extract implicit important words, then, organize the initial identifiers together with the freshly extracted words in a WordNet-like lexical view.

■ Sridhara *et al.* [Sridhara *et al.*, 2010] presented a novel technique to automatically generate comments for Java methods. They used the signature and the body of a method (*i.e.* method call) to generate a descriptive natural language summary of the method. The developer is left in charge to verify the accuracy of generated summaries. The objective of this approach is to ease developers' program comprehension. They used natural language processing techniques to automatically generate leading method comments. Studies have shown that good comments can help programmers quickly understand what a method does, assisting program comprehension and software maintenance.

They summarize the main actions of an arbitrary Java method by exploiting both structural and linguistic clues in the method. In their work, they focus on comments that describe a method's intent (*i.e.* descriptive comments) and other types. Descriptive comments summarize the major algorithmic actions of the method.

The process starts with method M as input. The approach extracts the method signature and method body such as method calls. Before any names can be analyzed for text generation, identifiers must be split into component words. They use camel-case splitting, which splits words based on capital letters, underscores, and numbers. Then they construct software word usage model to capture the action, theme, and secondary arguments for a given method along with its program structure. Then, they select the most important statements for the summary. After that they generate phrases for the selected statements and combine phrases to generate leading comments that summarize the main actions of Java method (*i.e.* summary comment for method M).

3.3.2.2 Code-To-Document Traceability Link Approaches

Here, we present code-to-document traceability link approaches. The documents represent use-cases or requirements of a single software system. The goal of these approaches is to facilitate the software comprehension. These approaches are different from code-to-feature traceability link approaches that aim to re-engineering software variants into SPL for the systematic reuse. Then we propose a synthesis of these approaches in Section 3.3.2.4.

● Grechanik *et al.* [Grechanik *et al.*, 2007] proposed a novel approach for automating part of the process of recovering traceability links (TLs) between types and variables in Java programs and elements of use-case diagrams (UCDs). The authors evaluated their prototype implementation on open-source and commercial software, and their results suggested that their approach can recover many traceability links with a high degree of automation and precision. UCDs are widely used to describe requirements and desired functionality of software products. These traces help programmers to understand code that they maintain and evolve.

They propose LeanArt approach that combines program analysis, run-time monitoring, and machine learning to automatically propagate a small set of initial TLs, between program variables and types (program entities such as class name, attribute name, method name, parameter name, *etc.*) and elements of UCDs. The input to LeanArt is software source code and UCDs. The core idea of LeanArt is that after programmers initially link a few program entities to elements of the UCDs, the system will glean enough information from these links to recover TLs for much of the rest of the program automatically. They implemented their technique as an Eclipse plug-in. The characteristic of LeanArt is to select a source, and then it displays targets linked to this source. The user can navigate from use cases to the related parts of the source code and vice-versa.

TLs bridge a gap between high-level concepts represented by elements of UCDs and low-level implementation details such as program entities. UCDs will be used later in our approach to documenting the mined feature implementation by giving names and description based on the use-case name and its description.

● Marcus *et al.* [Marcus and Maletic, 2003] use LSI to recover traceability links between source code and documentation. The input data consists of the source code (internal documentation) and external documentation. The external documentation is in the form of requirement documents which describe elements of the problem domain such as manual, design documentation, requirement documents or test suites. The requirement documents are supposed to have been written before implementation and do not include any parts of the source code.

They recover traceability links between high-level documents (requirements) and low-level documents (source code). IR techniques assume that all software artifacts are/can be put in some textual format. Then, they compute the textual similarity between each two software artifacts (the source code of a class and a requirement). A high textual similarity means that the two artifacts probably share several concepts and that, therefore, they are likely linked to one another.

In order to recover traceability links between source code and documentation the corpus must be ready. They form the text corpus out of both documentation (requirements) and source code (identifier names and comments). The identifier names in the source code are split into parts based simply on well-known coding standards such as camel-case. In that way they compute directly the similarity between external documents and source files. Each similarity which is higher than 0.70 is considered as a recovered link between source and documentation.

● Diaz *et al.* [Diaz *et al.*, 2013] propose an approach to capture relationships between source code artifacts for improving the recovery of traceability links between documentation and source code. They extract the author of each source code component and for each author they identify the "context" she/he worked on. Thus, for a given query from the external documentation (*i.e.* use case) they compute the similarity between it and the context of the authors. When retrieving classes that relate to a specific query using a standard IR-based approach (*e.g.* LSI or VSM) they reward all the classes developed by the authors having their context most similar to the query, by boosting their similarity to the query.

They proposed TYRION approach (*i.e.* semi-automated approach) to recovery of traceability links between use cases and Java classes of single software. The results indicate that code ownership information can be used to improve the accuracy of an IR-based traceability link recovery technique. IR-based traceability recovery process starts by indexing the artifacts in the artifact corpus through the extraction of terms from their content. The indexing process of the artifacts and the construction of the artifact corpus are preceded by a text normalization phase that aimed at: pruning out white spaces or non-textual tokens and splitting source code identifiers composed of two or more words into their constituent words.

The output of the indexing process is represented by a $m \times n$ matrix (*i.e.* term-document matrix), where m is the number of all terms that occur within the artifacts, and n is the number of artifacts in the repository. Once the artifacts are indexed, different IR methods can be used to compare a set of source artifacts (*i.e.* use cases) against another set of artifacts (*i.e.* source code files) and rank the similarity of all possible pairs of artifacts. For them use cases represent "queries" and source code files "documents". Once the list of candidate links has been generated, it is provided to the software engineer for examination. The software engineer reviews the candidate links, determines those that are correct links and discards the false positives.

3.3.2.3 Feature Documentation Approaches

We present here the relevant approaches to feature documentation. Feature documentation is not limited to source code only. We present two semi-automatic approaches documenting the identified feature implementations. In addition, we present an automatic approach documenting the identified features from product descriptions. Then we propose a synthesis of these approaches in Section 3.3.2.4.

❖ Davril *et al.* [Davril *et al.*, 2013] present an approach to constructing FMs from product descriptions. The task of extracting FMs from informal data sources includes mining feature descriptions from sets of informal product descriptions, naming the features in a way that is understandable to human users, and then discovering relations between features in order to organize them hierarchically into an inclusive model.

For purposes of building an FM, the clusters that represent features will be presented to human users. Those features must have meaningful names. The authors develop cluster-naming process that involved selecting the most frequently occurring phrase from among all of the feature descriptors in the cluster. To identify the most frequently occurring phrase they use the Stanford Part-of-Speech (POS) tagger to tag each term in the descriptors with its POS. The descriptors are then pruned to retain only nouns, adjectives, and verbs as the other terms were found not to add useful information for describing a feature. Frequent item-sets are then discovered for each of the clusters. In their context, frequent item-sets are sets of terms that frequently co-occur together in the descriptors assigned to the same cluster.

To select the name for a cluster, all of its frequent item-sets of maximum size, FIS_{max} are selected. Next, all feature descriptors in the cluster are examined. In each feature descriptor, the shortest sequence of terms which contains all the words in FIS_{max} is selected as a candidate name. For example, let FIS_{max} = {prevents, intrusion, hacker}. For a given feature descriptor such as: "prevents possible intrusions or attacks by hackers trying to enter your computer", the selected candidate name is "prevents possible intrusions or attacks by hackers". Finally, the shortest candidate name is selected, as this reduces the verbosity of the feature name. This work is discussed in Section 3.3.3.1.

❖ Ziadi *et al.* [Ziadi *et al.*, 2012] proposed semi-atomic approach to identify feature from object-oriented source code. Their approach takes as input the source code of a set of product variants. In their work they propose *manually* creating the feature names. In their approach they rely on the feature names that included in the original FM to name the identified feature implementations. For example, they manually named the identified feature implementation of sequence diagram as a *sequence diagram* feature. This work is discussed in Section 3.3.1.1.

❖ Yang *et al.* [Yang *et al.*, 2009] analyzed open source applications for multiple existing domain applications with similar functionalities. They propose an approach to recover domain feature models using data access semantics, FCA, concept pruning/merging, structure reconstruction and variability analysis. After concept pruning/merging, the analysts examine each of the generated candidate features (*i.e.* concept clusters) to evaluate its business meanings. Meaningless candidate features are removed, whilst meaningful candidate features are chosen as domain features. Then the analysts can name each domain feature with the help of the corresponding concept intent and extent. After this manual examination and naming, all the domain features are determined with significant names denoting their business functions. In their approach, they are *manually* naming the features based on the domain experts. This work is discussed in Section 3.3.3.2.

3.3.2.4 Synthesis

Here, we summarize the related work for source code documentation according to the criteria presented at the beginning of this section. Table 3.4 presents a comparison between source code documentation studies. In the following, we mention high level remarks on the related work from Table 3.4.

❏ The majority of existing approaches are designed to extract labels, names, topics or code summarization in a single software system.

❏ The majority of existing approaches manually assign feature names to the feature implementations.

❏ Some approaches identify code-to-document traceability link in single software to facilitate code comprehension.

❏ LSI is the most used technique in source code comprehension. Static analysis of source code is the most used type in the software comprehension.

❏ There is no study tried to documenting the identified feature implementations from the source code of software variants based on available artifacts (*e.g.* use-case diagrams).

❏ There is no study tried to documenting the identified feature implementations by using source code itself (*i.e.* identifier names).

❏ The majority of existing approaches are designed to extract labels, names, topics based on the class names of single software. The most used input is class.

❏ The identifier names such as package, class, attribute or method considered as the most important elements to understand existing software system.

❏ In order to document the mined feature implementations from the source code of software variants; we rely on the use-case diagrams of software variants or source code element names for each implementation. Linking the use-cases to feature implementations is actually no different than recovering traceability links between them in one version of the software. The novelty of our way is that we exploit commonality and variability across software variants, at feature implementations and use-cases levels, to apply the LSI method in an efficient way.

Table 3.4 : Summary of source code documentation studies (comparison table).

ID	Reference	Objectives								Software		Programmed method		Code analysis			Inputs											Techniques												Outputs										Language	Case study	Tool support
		Label software components	Label software features	Extract code topics	Extract code terms	Extract code summarization	Extract structure of software	Generate comments	Traceability link	Single software	Software family	Automatic	Semi-automatic	Static	Dynamic	UML class diagram	Package	Class	Attribute	Method	Method body	Components	Use-case diagram	Use-cases	Documents	Product descriptions	Software entities/identifiers	Automatic heuristic	Log-likelihood ratio	Part-of-speech	Data access semantics	FCA	LSI	VSM	Code dependencies	Ad hoc algorithm	Natural language processing	Machine learning	Run-time monitoring	Component name	Labels	Topics	Terms	Summarization	Feature name	Feature name/ manually	WordNet-like lexical view	Comments	Traceability link	Java		
1	[Kebir *et al.*, 2012]	✗								✗		✗		✗								✗						✗												✗										✗	✗	
2	[Kuhn, 2009]	✗								✗		✗		✗								✗							✗												✗									✗	✗	✗
3	[Kuhn *et al.*, 2007]			✗						✗			✗	✗				✗															✗									✗								✗	✗	✗
4	[Lucia *et al.*, 2012]				✗					✗			✗	✗				✗															✗	✗									✗							✗	✗	✗
5	[Haiduc *et al.*, 2010]					✗				✗		✗		✗			✗	✗		✗													✗		✗									✗						✗	✗	✗
6	[Falleri *et al.*, 2010]						✗			✗		✗		✗													✗										✗										✗			✗	✗	
7	[Grechanik *et al.*, 2007]								✗	✗			✗	✗									✗				✗											✗	✗										✗	✗	✗	
8	[Marcus and Maletic, 2003]								✗	✗			✗	✗											✗		✗						✗																✗	✗	✗	✗
9	[Diaz *et al.*, 2013]								✗	✗			✗	✗				✗						✗									✗	✗															✗	✗	✗	✗
10	[Sridhara *et al.*, 2010]							✗		✗		✗		✗						✗	✗															✗												✗		✗	✗	
11	[Davril *et al.*, 2013]		✗								✗	✗														✗				✗															✗						✗	✗
12	[Ziadi *et al.*, 2012]		✗								✗		✗			✗	✗	✗	✗	✗																✗										✗				✗	✗	
13	[Yang *et al.*, 2009]		✗								✗		✗	✗	✗					✗											✗	✗														✗				✗	✗	✗

3.3.3 Reverse Engineering FMs Approaches

In this section, we present the FM extraction approaches from different software artifacts such as requirements, descriptions, configurations and source code. In literature there are many approaches related to FM extraction from different artifacts; we select the closest approaches to our work. We classify the FM extraction approaches into two categories: approaches dealing with high level models and those dealing with source code. In Section 3.3.3.1, we present the reverse engineering FMs approaches from high level models. Section 3.3.3.2 presents the reverse engineering FMs approaches from source code. Section 3.3.3.3 concludes reverse engineering FMs approaches. It presents a synthesis and comparison table for these approaches. We present the selected papers in Table 3.5.

Table 3.5 : Summary of reverse engineering FMs studies (the selected papers).

ID	Reference	Reverse Engineering FMs	
		High level model	Source code
1	[Davril *et al.*, 2013]	✗	
2	[Ryssel *et al.*, 2011]	✗	
3	[She *et al.*, 2011]	✗	
4	[Acher *et al.*, 2012]	✗	
5	[Acher *et al.*, 2013b]	✗	
6	[Chen *et al.*, 2005]	✗	
7	[Haslinger *et al.*, 2011]	✗	
8	[Loesch and Ploedereder, 2007]	✗	
9	[Braganca and Machado, 2007]	✗	
10	[Yang *et al.*, 2009]		✗
11	[Paškevičius *et al.*, 2012]		✗

We evaluate the studied works according to the following *criteria*: objectives of the study (re-engineering, restructuring variability of SPL, domain analysis, understanding, *etc.*), target software (single software versus software family), programmed method (automatic versus semi-automatic), inputs (product configurations, classes, methods, use-case diagram, incidence matrix, product descriptions, variability descriptions, requirements, *etc.*), techniques (text similarity, FCA, *etc.*), outputs (FMs, variability classification, CTCs, and-group, xor-group, or-group, *etc.*), target languages (*e.g.* Java), types of evaluation (case study or empirical data) and tool support.

3.3.3.1 Reverse engineering FMs from high level models

We present here a collection of relevant approaches to the reverse engineering of FMs from high level models. We present each approach separately. Then we propose a synthesis of these approaches in Section 3.3.3.3.

■ Davril *et al.* [Davril *et al.*, 2013] present a novel, automated approach for constructing FMs from publicly available product descriptions found in online product repositories and marketing websites. Each individual product description provides only a partial view of features in the domain; a large set of descriptions can provide fairly comprehensive coverage. Their approach utilizes hundreds of partial product descriptions to construct an FM. They focus on the scenario in which an organization has no existing product descriptions and must rely upon publicly available data from websites. However, such product descriptions are generally incomplete, and features are described informally using natural language.

The task of extracting FMs from informal data sources involves mining feature descriptions from sets of informal product descriptions, naming the features in a way that is understandable to human users, and then discovering relationships between features in order to organize them hierarchically into a comprehensive model. In their approach, product specifications are mined from online software repositories. Then, these product specifications are processed in order to identify a set of features and to generate a *product-by-feature matrix* ($P \times F$) in which the rows of the matrix correspond to products and the columns correspond to features. Then, meaningful names are selected for the mined features. Next, a set of association rules are mined for the features. These association rules are used to generate an implication graph (IG) which captures binary configuration constraints between features. The tree hierarchy and then the feature diagram are generated given the IG and the content of the features. Finally, *cross-tree constraints* and *OR-groups* of features are identified.

To evaluate the quality of the generated FMs, they first explored the possibility of creating a "*golden answer set*" and then comparing the mined FM against this standard. The results show that the FM generated by their approach does not reach the same level of quality achieved in manually constructed FMs. The evaluating of the quality of the generated FM is time-consuming and involves manually creating one or more FMs for the domain, and then asking human users to evaluate the quality of the product lines in a blind study.

■ Ryssel *et al.* [Ryssel *et al.*, 2011] present an approach based on FCA that analyzes incidence matrices containing matching relations as input and creates FMs as output. The resulting FMs describe exactly the given input variants. In their approach the extracted FM consists of group of feature constraints without cross-tree constraints. In their context, there are neither preexisting FMs nor propositional formulas. They exploit FCA to synthesize FMs including or- and xor groups. Their synthesis technique is based on FCA and translates feature group recovery to minimal set cover problems.

For the tree recovery stage, first, they recover a Directed Acyclic Graph (DAG) that represents all possible tree hierarchies given the input. The DAG is recovered as an attribute concept graph by checking subsets of features between configurations. Feature groups are found by solving minimal set cover problems. The possible set cover candidates consist of only immediate children of a given parent. They also address the recovery of complex implications. They first build an extended attribute concept graph that includes negated features as nodes. The complex implications are identified by solving a minimal set cover problem using the extended attribute concept graph to limit the problem size.

In their work they construct concept lattice which is used to derive the feature hierarchy. They create feature models, which specify exactly the variants given as input in the form of an incidence matrix. Using optimized methods based on FCA, the feature models are generated in very short time, because they scale significantly better than the standard methods of FCA to calculate the lattices and implication bases. Their work doesn't support the mandatory features and cross-tree constraints (*e.g.* require/exclude). In addition, there are many missing features in the extracted FMs.

■ She *et al.* [She *et al.*, 2011] present procedures for reverse engineering FMs based on a crucial heuristic for identifying parents which are the major challenge of this task. They also automatically recover constructs such as feature groups, mandatory features, and implies/excludes edges. They evaluate the technique on two large-scale SPLs with existing reference FMs. Their approach combines two distinct sources of information: textual feature descriptions and feature dependencies as propositional formula (*i.e.* inputs).

They developed an efficient synthesis procedure to compute variability information (*e.g.* feature groups) and proposed heuristics for identifying the most likely parent feature candidates of each feature based on text similarity and domain knowledge. Their heuristics are specific to the targeted systems (Linux, FreeBSD, eCos both from the domain of operating system). Their procedure does not detect *Or-groups*. The algorithm takes a set of features, dependencies, and feature descriptions to present potential parents for feature to help a modeler build the feature hierarchy. Given a feature, the algorithm creates a list containing the implied features ranked by their textual similarity, and a second list containing all features ranked by their textual similarity for situations where input dependencies may be missing. This algorithm was the first to use both logical dependencies and textual similarity heuristics for FM synthesis.

■ Acher *et al.* [Acher *et al.*, 2012] present a procedure (*i.e.* semi-automatic procedure) to synthesize FM based on the product descriptions. Their approach takes as input product description for a collection of product variants to build the FM. Products are described by characteristics (*e.g.* language, license, *etc.*) with different patterns on values (*e.g.* many-valued, one-valued, *etc.*). Product descriptions are interpreted to build as much FMs as there are products. Finally, the FMs of the products are merged, producing a new FM that compactly represents valid combinations of features supported by the set of products.

Their technique exploits structurally similar product descriptions and uses a conversion specification to describe the transformation of a single semi-structured product description to a FM. The individual FMs are then merged to form a final FM describing all products in the dataset. The conversion specification is tailored to parse and interpret the specific structure of the input data. The individual FMs share the same hierarchy making the final merge relatively simple. They rely on their FAMILIAR domain specific language for performing the merge operations. Their heuristics exploit the structure of the data to automatically select a hierarchy.

In their work semi-structured product descriptions are used as input. The product descriptions are similar to feature configurations; however, product descriptions can contain variability. For example, a product could support one or more storage methods, or exactly one operating system. Each product description is transformed into a product FM that describes the specific description and its variability. The resulting set of product FMs are then merged to create a FM describing variability in all products. They used product descriptions ranging from 9 to 190 features as input.

■ Acher *et al.* [Acher *et al.*, 2013b] propose a reverse engineering process for producing a variability model of a plugin-based architecture. They present a comprehensive, tool-supported process for reverse engineering and evolving architectural FMs. They develop automated techniques to extract and combine different variability descriptions, including a hierarchical software architecture model, a plugin dependency model and the software architect knowledge (*i.e.* several inputs). The basic idea is that variability and technical constraints of the plugin dependencies are projected onto an architectural model. After the extraction, alignment and reasoning techniques are applied to integrate the architect knowledge and support the extracted FM. This approach has been applied to a representative, large scale plugin-based system (FraSCAti), considering different versions of its architecture.

They presented a tool-supported approach to extract and manage the evolution of software variability from an architectural perspective. The process involves the automatically supported extraction, aggregation, alignment and slicing of architectural FMs. They use several sources of information as inputs. As a result, they contribute to projecting the implementation constraints (expressed by plugin dependencies) onto an architectural model. The process enables the software architect to validate the extracted FMs. The authors observed that the retained hierarchy and feature groups can be inappropriate.

To summarize the work, their goal is reverse engineering architectural FMs (domain: component and plugin based systems, stakeholder: software architect). Multiple variability sources (including architect knowledge) are combined to construct semi-automatically an FM (*i.e.* concern). The constructed FM consists of 92 features and 158 constraints (*i.e.* output/complexity). In their work they rely on the reconciling FMs technique; where two (or more than two) FMs cannot be directly compared or merged: They are reconciled by removing unnecessary details they used slice and merge operators.

■ Chen *et al.* [Chen *et al.*, 2005] propose a requirements clustering-based approach (*i.e.* semi-automatic approach) to constructing FMs from the functional requirements of sample applications. In their approach, sample applications are first analyzed, and a set of corresponding application FMs are built. Then these application FMs are merged into a domain FM and features are manually labeled based on the difference analysis of these application FMs. The proposed approach consists of two steps: construction of Application Feature Trees (AFTs) and construction of Domain Feature Tree (DFT).

In the first step, several sample applications are analyzed. For each of them there are three activities. The first activity is to elicit functional requirements for the sample application. They assume each of the functional requirements is documented as long as it clearly specifies a certain behavior of the system. These requirements are called individual requirements in their work. The second activity is to model individual requirements and the relationships between them in an undirected graph, which is called Requirements Relationship Graph (RRG). The third activity is to identify and organize features by applying the clustering algorithm in RRG. The underlying idea is that a feature is a cluster of closest related requirements, and features with different granularities can be generated by changing the clustering threshold value. After finishing these three activities, they get the application feature tree for each application.

In the second step, they merge the application feature trees of all the sample applications into a single domain feature tree (FM), and label the features. The extracted domain FM consists of optional and mandatory features without any constraints or until group of features. The goal of the clustering is to synthesize an individual FM for each requirements document where each document describes a single variant. In terms of the scenario's size, they used requirements from two sample applications where one application had 21 requirements. In addition they apply clustering on manually constructed RRGs to create FMs. The synthesis technique requires that weighted graphs describing relations between requirements be built by manually analyzing requirements documents. The weighted graphs are then clustered to identify similar features and construct the feature hierarchy. In their work, clustering is based on edge weights instead of textual similarity. They rely on the user for marking features as optional and mandatory features.

■ Haslinger *et al.* [Haslinger *et al.*, 2011] present a fully automatic algorithm that reverse engineers a basic FM (*i.e.* FM without cross-tree constraints) from the feature sets which describe the features each system provides. The extracted features are directly connected to the root feature. Their approach accepts as input tabular data which includes the product configurations and produces basic FM.

They propose an automated algorithm for extracting a distinct feature tree given the input configurations. In their algorithm, they arbitrarily select a feature from a feature list to be the root of the FM (*i.e.* first feature in the table). Then, all mandatory features are connected to the root features. The optional features classify into or- and xor groups or connect directly to the root features based on the configurations. In their work optional features that always appear together in all products are detected as atomic sets of features.

They do not address general FMs that can contain any type of cross-tree constraints. In their work only one FM can be reversed engineered (*i.e.* unique FM). Nevertheless, the problem is that the resulting FM may differ from the expectation of the user. In their evaluation, the authors state that the synthesized model may not match the input model; however, the models are equivalent in terms of their configurations. In case the features order in the input configurations was changed all results will change. The algorithm proposed do not control the way the feature hierarchy is synthesized in the resulting FM and may generate FM whose configurations violate the input dependencies.

■ Loesch *et al.* [Loesch and Ploedereder, 2007] present a method for restructuring and simplifying the provided variability in an SPL based on FCA (*i.e.* without feature and FMs extraction). They discussed a variability extraction and visualization method based on concept analysis. Their work does not use or produce FMs.

They present a new method to restructure and simplify the provided variability in a SPL. Their method is based on FCA. First, a table is constructed that reflects the usage of variable features in product configurations. From the table, a concept lattice is derived that factors out which variable features are commonly used in product configurations and which variable features only appear in specific product configurations. Using the sparse representation of the lattice, they classify the usage of variable features in product configurations into: ❶ always used, ❷ never used, ❸ only used mutually exclusively and ❹ only used in pairs.

Variable features appearing at the top concept in the lattice are used in every product configuration. These features are likely to be mandatory features (*i.e.* always used). Variable features appearing at the bottom concept are not used in any of the product configurations (*i.e.* never used). For two variable features F1 and F2 that appear at different concepts and whose infimum is the bottom concept (*i.e.* F1 is introduced in concept C1, F2 is introduced in concept C2 and $C1 \sqcap C2 = \bot$), they are only used mutually exclusively in the product configurations. These features are likely to be alternative features (*i.e.* only used mutually exclusively). Based on the concept lattice if two variable features F1 and F2 that appear at the same concept, the two features cannot be used separately (*i.e.* only used in pairs). Require constraint (F1 requires F2) can be extracted from the lattice via background implications (*i.e.* upward paths in the lattice). For exclude constraint they rely on mutually exclusively relation to extract this type of constraints.

■ Braganca *et al.* [Braganca and Machado, 2007] propose an approach to automate the transformation from UML use-case diagram to FMs. Their approach explores include and extend relationships between use-cases to discover relationships between features. In their work, each functional use-case is mapped to a feature. They establish a model-driven approach to deriving a concrete use-case diagram that represents one product of a product line based on the feature configuration.

Basically, their approach consists of three transformations: transform a family use-case model into FM; transform FM into a configuration metamodel (Ecore model); and finally, transform a configuration model and a family use-case model into an application use-case model. The approach accepts as input use-case diagrams and produces FMs. They consider the functional use-cases. They present a prototype based on the Eclipse Modeling Framework (EMF) and SmartQVT. The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. SmartQVT is a tool set for model-to-model transformations that implements the QVT relations language in a Java language.

To summarize the work, they present a model-driven approach to map use-cases to features. Each use-case is mapped to a feature. Top use-case becomes root feature. The complete structure of the FM can only be created by examining the relations between use-cases. In their approach we cannot say a use case is mandatory or optional without a context. This context results from the relationships the use-case has with other use-cases. For instance, if the functionality of a use-case is always referenced by other use-cases, then we can say that such a use-case is mandatory.

3.3.3.2 Reverse engineering FMs from source code

We present here a collection of relevant approaches to the reverse engineering of FMs from source code. We present each approach separately. Then we propose a synthesis of these approaches in Section 3.3.3.3.

❖ Yang *et al.* [Yang *et al.*, 2009] analyzed open source applications for multiple existing domain applications with similar functionalities. They propose an approach to recover domain feature models using data access semantics, FCA, concept pruning/merging, structure reconstruction and variability analysis. The basic assumption in their approach is that data models (entity-relationship) of these applications are similar and uncover a basis for mapping the common concepts among different applications. The starting point of the mining is a reference domain data model: domain experts construct a domain data model and establish mappings from application data schemas to the domain data model. The result of this step is data schema mappings.

The data schema of the applications is then mapped to the reference domain model by detecting entities or fields with similar names in the applications. The approach is based on detecting consistent data access semantics (*i.e.* similar usage of data entities by the methods in the applications). The data access semantics are obtained by detecting SQL statements using aspects which intercept invocations to the database library and store SQL run-time records. In order to verify that different methods have a similar data access semantics, the SQL records must contain the tables, fields and constraints involved in the queries. These records describe the data access semantics of each method in each application.

Based on the data access semantics of all the methods, they conduct FCA (formal context consists of all the methods as objects and their data access semantics as attributes). The result of this step is the original concept lattice. The records are then analyzed using FCA. The resulting concepts represent the usage of data-entities mapped to the reference domain model. Depending on the level of the concept, it is merged with neighbor concepts or pruned so that each concept represents one feature. The result is a domain feature diagram comprising mandatory features, optional features, whether they are alternative or multiple, its variants, and whether they are inclusive or exclusive. The output of their approach is limited to basic FM without cross-tree constraints (requires and excludes). However, variability dependencies (*e.g.* feature A requires or excludes feature B) can be obtained from the concept lattice.

❖ Paškevičius *et al.* [Paškevičius *et al.*, 2012] present a framework for the automated derivation of FMs from the existing software artefacts (*e.g.* classes, components, libraries, etc.), which includes a formal description of FM, a program-feature relation meta-model, and a method for FM generation based on feature dependency extraction and clustering. FMs are generated in Feature Description Language (FDL) and as Prolog rules. They focus on reverse engineering of source code to FMs. However, there is a wide gap between FMs and program source code.

The novelty of this paper is a proposed method for the automatic derivation of FMs from Java source code of *single software*. Their approach accepts as input a set of classes of single software and extracts basic FM. The extracted FM does not include require/exclude constraints. In their work each *class* in software represents a *feature*. Feature (class) dependencies are extracted by parsing Java class files. They use the dependency finder tool[1] to parse, analyze and extract dependencies from Java class files. Their methodology of FM extraction is as follows: compile Java source code using a standard Java compiler, after that, they extract feature dependencies from Java class files, then, they construct a feature distance matrix (*i.e.* dependencies between features), after this step, they cluster features based on their dependency in a feature tree, based on this step, they convert a feature tree into FM, lastly, they generate a description of FM in FDL/Prolog.

3.3.3.3 Synthesis

Here, we summarize the related work for reverse engineering FMs according to the criteria presented at the beginning of this section. Table 3.6 presents a comparison between reverse engineering FMs studies. In the following, we mention high level remarks on the related work from Table 3.6.

❏ The majority of existing approaches are designed to reverse engineering FM from high level models (*e.g.* product description and requirements) and other approaches deal directly with low level model (source code) with a lot of limitations.

❏ Few works extract basic FM from source code. These approaches extract FM from single software or from software family. All approaches are working with a class or method granularity. The resulting FM consists of basic elements without cross-tree constraints.

❏ Some approaches offer a solution acceptable but not able to identify important parts of feature model such as cross-tree constraints, and-group, or-group, xor-group, parent features.

[1] http://depfind.sourceforge.net/

- The main challenge of works that reverse engineering FMs from product configurations is that numerous candidate FMs can be extracted from the same input configurations, yet only a few of them are meaningful and maintainable.

- Techniques for synthesizing an FM from a set of dependencies (*e.g.* encoded as a propositional formula) or product configurations/descriptions have been proposed in literature such as [She *et al.*, 2011] [Haslinger *et al.*, 2011] [Acher *et al.*, 2012]. An important limitation of prior works is the identification of the feature hierarchy (*resp.* parent features (*i.e.* group of features)) when synthesizing the FM.

- In our work, we rely on FCA to extract FMs from the source code of software variants. Two approaches [Yang *et al.*, 2009] [Ryssel *et al.*, 2011] rely on FCA to extract basic FM without cross-tree constraints. Another approach [Loesch and Ploedereder, 2007] uses FCA as a tool to understand the variability of existing SPL based on product configurations. All these approaches share the usage of a single form of input to extract variability information.

- From the inputs we can note that FMs can be used in different phases of the SPL development, from high-level requirements to code implementation. FMs can be applied to any kind of artifacts and at any level of abstraction.

- The majority of existing approaches are designed to identify the dependencies between features regardless of FM hierarchy.

Table 3.6 : Summary of reverse engineering FMs studies (comparison table).

ID	Reference	Objectives					Programmed method		Inputs										Techniques						Outputs							Software		Language	Tool Support	Evaluation	
		Restructuring variability of SPL	Transform use-case diagrams to FMs	re-engineering	Domain analysis	Understanding and generative approaches	Automatic	Semi-automatic	Product configurations	Classes	Methods	Use-case diagram	Incidence matrix	Product descriptions	Feature dependencies	Feature descriptions	Variability descriptions	Requirements	Transformation processes	Ad hoc algorithm	Text similarity	FCA	User input and heuristics	Weighted graph clustering	Variability classification	Unique FM	Many FMs	CTCs	And-group	Xor-group	Or-group	Single software	Software family	Java		Case study	Empirical data
1	[Davril *et al.*, 2013]				✗			✗						✗						✗							✗	✗			✗		✗		✗	✗	
2	[Ryssel *et al.*, 2011]			✗			✗						✗							✗		✗				✗				✗	✗		✗		✗	✗	
3	[She *et al.*, 2011]					✗		✗							✗	✗				✗	✗					✗		✗	✗	✗			✗			✗	
4	[Acher *et al.*, 2012]				✗			✗						✗									✗				✗	✗	✗	✗	✗		✗		✗	✗	
5	[Acher *et al.*, 2013b]			✗			✗										✗			✗							✗	✗	✗	✗	✗		✗		✗	✗	
6	[Chen *et al.*, 2005]			✗				✗										✗						✗		✗							✗			✗	
7	[Haslinger *et al.*, 2011]			✗			✗		✗											✗							✗			✗	✗		✗		✗	✗	
8	[Loesch and Ploedereder, 2007]	✗					✗		✗													✗			✗			✗		✗	✗		✗		✗		✗
9	[Braganca and Machado, 2007]		✗					✗				✗							✗							✗		✗					✗		✗	✗	
10	[Yang *et al.*, 2009]				✗			✗			✗											✗				✗			✗	✗	✗		✗	✗	✗	✗	
11	[Paškevičius *et al.*, 2012]				✗		✗			✗										✗						✗			✗	✗		✗		✗	✗	✗	

3.4 Summary

In this chapter, we survey the different approaches related to the feature location, source code documentation and reverse engineering FMs field. We present our observations regarding the studied approaches which will be useful to introduce our contributions. We aim at providing an approach based on several techniques (*i.e.* FCA, RCA and LSI), in order to contribute in providing a solution for reengineering software variants into a SPL. In this chapter, we presented several concepts and works relevant to our approach. This chapter provides a concise overview of the different approaches and shows the need to propose *REVPLINE* approach. We are interested by three main problems: ① Feature location, ② Feature documentation and ③ Reverse engineering FM from object-oriented source code of a collection of software product variants.

Concerning the *first* problem, we have listed a number of works that are based on different techniques to identify source code elements that implements specific functionality. Here, we highlight the main limitations of these works: ❶ The majority of existing approaches are designed to locate the program elements of a particular feature in a *single software system*; ❷ Some existing work accepts as input *two* sources of information such as source code of software variants and feature/feature description; ❸ Some existing approaches investigate variability at package or class levels. These approaches fail in case that software variability is represented mainly at the method body level for example; ❹ Some works extract common block and sets of variable blocks from software variants source code. These works are limited to this only, and they do not distinguish between features within the same block such as mandatory features in the common block.

REVPLINE separates the source code element set of software variants in two subsets, the common features set and the optional features set. Then, REVPLINE separates the optional feature set into small subsets such that each contains source code elements shared by groups of two or more variants or unique for single variant. Then, for each subset, REVPLINE splits the source code elements into a set of blocks based on the lexical and structural similarity, where each block represents feature implementation.

Concerning the *second* problem, we have listed a number of works that are based on different techniques to document the source code. Here, we highlight the main limitations of these works: ❶ The majority of existing approaches are designed to extract labels, names, topics or code summarization in a single software system; ❷ The majority of existing approaches manually assign feature names to the feature implementations. In the literature there is no work which gives a name or description for the mined feature implementation.

REVPLINE identifies a set of feature implementations as source code elements. To exploit the mined feature implementations and build the FM, feature implementations must be documented as a first step. REVPLINE documenting the mined feature implementations by giving names and descriptions, based on the feature implementations and use-case diagrams of software variants. The novelty of our approach is that we exploit commonality and variability across software variants, at feature implementations and use-cases levels, to apply information retrieval methods in an efficient way. REVPLINE assigns name and description for each feature implementation based on the use-case name/description. We rely on RCA, FCA and LSI techniques to document the mined feature implementations. In case of absence of use-case diagrams, we rely on source code element names to assign the name of each feature implementation.

Concerning the *third* problem, we have listed a number of works that are based on different techniques to reverse engineering FMs. Here, we highlight the main limitations of these works: ❶ The majority of existing approaches are designed to reverse engineering FM from high level models (*e.g.* product description and requirements) and other approaches deal directly with low level model (*i.e.* source code) with a lot of limitations; ❷ The main challenge of works that reverse engineering FMs from *product configurations* is that numerous candidate FMs can be extracted from the same input configurations, yet only a few of them are meaningful and maintainable; ❸ Some approaches offer a solution acceptable but not able to identify important parts of feature model such as cross-tree constraints, and-group, or-group, xor-group, parent features.

REVPLINE identifies a set of feature implementations as a set of source code elements of software variants and associate each feature with its name. There is need to synthesize a correct and consistent FMs. We rely on the mined and documented features to extract FMs based on FCA.

This chapter presents the most closely related work to our contributions. We provided a concise overview of the different approaches and show the need to propose REVPLINE approach. We presented at the end of each category of related work a *synthesis* and *comparison table* between the presented approaches. Figure 3.1 shows the basic elements of REVPLINE approach (built using FreeMind software[2]).

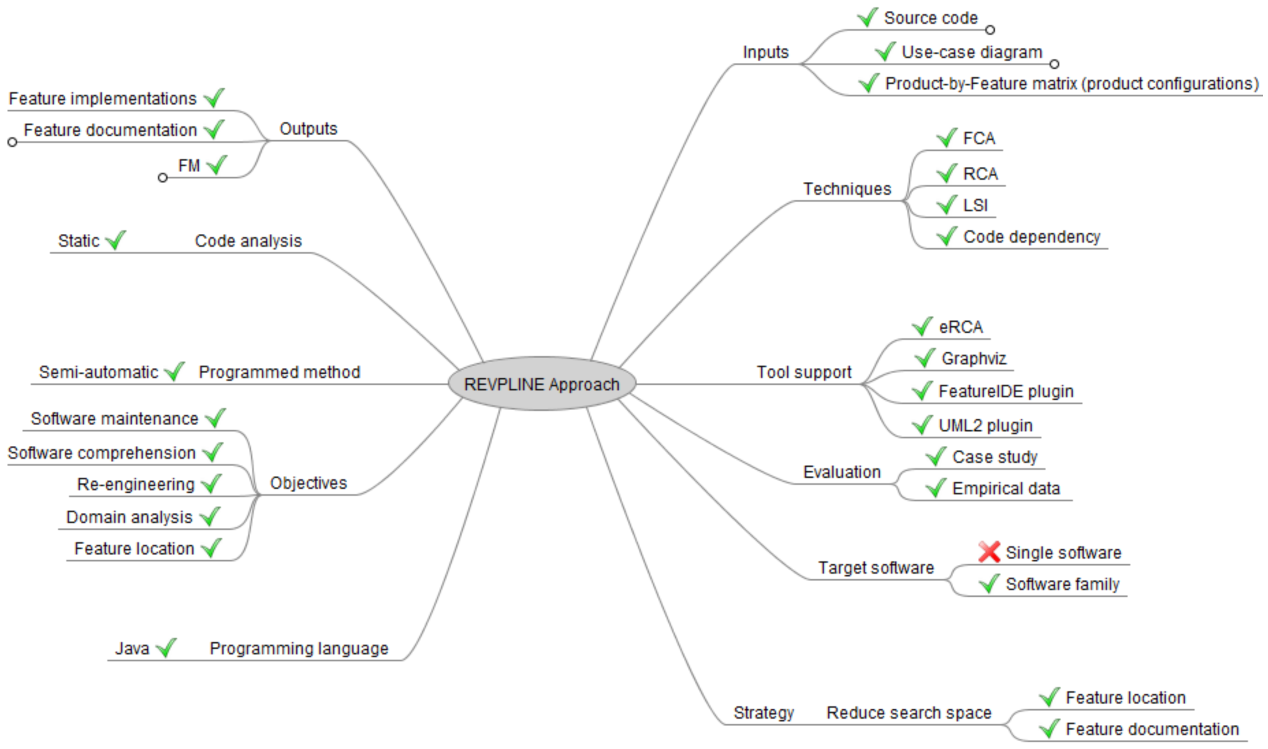


Figure 3.1 : The basic elements of REVPLINE approach.

[2]http://sourceforge.net/projects/freemind/files/

Part II

RIVEPLINE Approach: Contributions

CHAPTER 4

REVPLINE: Feature Location in a Collection of Software Product Variants

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Don't Repeat Yourself PRINCIPLE

Preamble

This chapter presents the first contribution of the REVPLINE approach. We introduce the feature mining process from object-oriented source code of a collection of software product variants. Section 4.1 gives a presentation of the problem. Section 4.2 develops the principles of the proposal. Section 4.3 presents the feature mining process. Next Section 4.4 discusses threats to the validity of the feature mining process. Finally in Section 4.5, we conclude this chapter.

4.1 Presentation of the Problem

Companies often develop a set of software variants that share some features and differ in other ones to meet specific requirements. REVPLINE approach aims to reverse engineering FM from source code of software variants. This is a main step to re-engineering software product variants into a SPL for systematic reuse. To build a FM, it is necessary to mine optional and mandatory features from the source code of the software variants. Thus, we propose, in this chapter, the first contribution of the REVPLINE approach which aims to mine features from the object-oriented source code of a set of software variants. The approach is based on FCA, LSI and code dependency.

This chapter proposes an approach to identify feature implementations from a collection of software product variants in order to define the FM of these variants. Our approach is based on reducing the search space of all source code elements of all variants by exploiting commonality and variability at source code level. Furthermore, we rely on the lexical and structural similarity between the source code elements for each reduced search space to further reduce it into atomic sets of source code elements which represent the feature implementations.

A SPL is usually characterized by two sets of features: the features that are shared by all products in the family, which represent the *SPL's commonalities* (*i.e.* mandatory features), and the features that are shared by some, but not all, products in the family, which represent the *SPL's variability* (*i.e.* optional features). SPLs are usually described with a *de-facto* standard formalism called *FM*. In common software development processes, software product variants often evolve from an initial product developed, for and successfully used by the first customer. These product variants usually share some common features but they are also different from one another, due to subsequent customization to meet the specific requirements of different customers [Xue *et al.*, 2012].

When variants become numerous, switching to a rigorous SPLE process is a solution to tame the increasing complexity of all the engineering tasks. To switch to SPLE starting from a collection of existing variants, the first step is to mine the FM that describes the SPL. This implies identifying the software family's common and variable features. Manual reverse engineering of the features/FM of software variants is time-consuming, error-prone, and requires substantial efforts [Ziadi *et al.*, 2012].

As we have shown in the state of the art section, the majority of existing approaches are designed to identify code-to-feature traceability link in a single software system such as [Marcus *et al.*, 2004] [Poshyvanyk and Marcus, 2007]. Other approaches are designed to identify code-to-feature traceability link in a collection of software product variants such as [Xue *et al.*, 2012] [Linsbauer *et al.*, 2013]. These approaches accept as input in addition to the source code itself feature names and their descriptions. These approaches linking each feature with the corresponding source code elements are based on lexical or structural similarity. Some existing approaches are designed to identify features in a collection of software variants based on the source code only, such as [Ziadi *et al.*, 2012] [Rubin and Chechik, 2012]. These approaches identify only the commonality block and variability blocks across software variants.

In our approach, we identify commonality block and variability blocks across software variants at all levels of source code elements not only at class level (*i.e.* all levels of variability). Furthermore, we reduce each block into a set of features based on the lexical and structural similarity. Our approach only accepts as input the source code of product variants; we don't know features in advance.

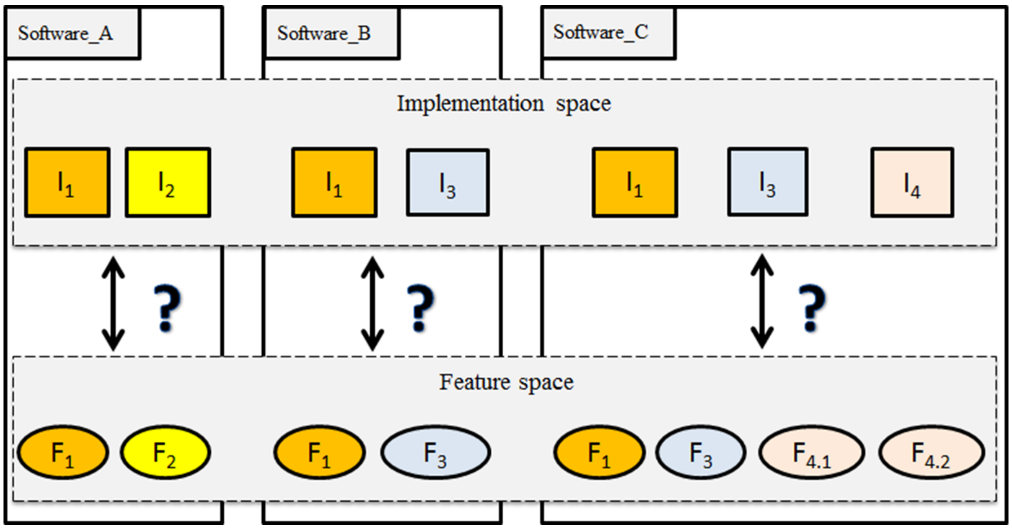


Figure 4.1 : Feature location in product variants (inspired by [Xue *et al.*, 2012]).

This chapter proposes an approach for *feature location in a collection of software product variants*. The approach is based on the identification of the implementation of software variant features among object-oriented source code elements (*cf.* Figure 4.1; where I_i are source code elements and F_j are features). These source code elements constitute the initial search space. We exploit commonality and variability across software variants at source code level to reduce this search space. We further use textual and code dependency to define a similarity measure that enables to identify subgroups of elements that characterize the implementation of each possible feature. Figure 4.5 gives an overview of our approach.

4.2 Principles of the Proposal

This section presents the main principles used in our approach for mining features from source code. It also shortly describes the example that illustrates the remaining of this chapter.

4.2.1 Goal and Core Assumptions

The general objective of our work is to mine the FM of a collection of software product variants based on the static analysis of their source code. Mining common and variable features is a first necessary step towards this objective.

We consider that "a feature is a prominent or distinctive and user visible aspect, quality, or characteristic of a software system or systems" [Kang, 1990]. Our work focuses on the mining of *functional* features. Functional features express the behaviour or the way users may interact with a product. We consider that a feature represents an aspect valuable to the customer. It is represented by a single term. We adhere to the classification given by [Kang, 1990] which distinguishes three categories of features: functional, operational, and presentation features. In our work, we focus on the functional features.

As there are several ways to implement features [Al-Msie'deen *et al.*, 2013a], we consider software systems in which functional features are implemented at the programming language level (*i.e.* source code). We also restrict to OO software. Thus, features are implemented using object-oriented building elements (OBEs) such as packages, classes, attributes, methods or method body elements (*e.g.* local variable, method invocation, attribute access). We consider that a feature corresponds to one and only one set of OBEs. This means that a feature always has the same implementation in all products where it is present. This assumption based on the way that software product variants are developed. In our work, we assume that software variants are developed by using *clone-and-own* approach (*i.e.* copy-paste-modify). The clone-and-own approach refers to the practice of creating a new software product by "cloning" a copy of the source code of another software product and then modifying it by add or remove features [Rubin and Chechik, 2013a] [Dubinsky *et al.*, 2013]. We also consider that feature implementations may overlap: a given OBE can be shared between several features' implementation (*i.e.* junction).

Definition 4.1. *Junction: "A junction is a collection of object-oriented building elements (such as class, method or attribute) that is common to two or more feature implementations".*

4.2.2 Object-oriented Source Code Variation

In our work, we focus on the following four levels of variations: ❶ package variation, ❷ class variation, ❸ attribute variation and ❹ method variation (*cf.* Figure 4.2).

Package variation encompasses variation at two levels: package set variation (set of packages that appear in some software variants but not all variants) and package content variation (this means all software variants have the same packages but with different contents (*e.g.* different classes)). *Class variation* shows variation on two levels: class signature variation and class content variation. Class signature variation means that two or more classes have the same name but declare different relations (*e.g.* implements, extends) or different access levels (*e.g.* public). Class content variation means that a class appears in many software variants but with different content (*e.g.* attributes and methods). *Attribute variation* can be found in attribute declarations, such as access level (protected, private and public) and data type (float, int, double, string, char, boolean, default values, *etc.*). *Method variation* can appears in the method signature (access level, returned data type and parameters list (name, data type and order)) and in the body of the method (local variable, method invocation and attribute access).

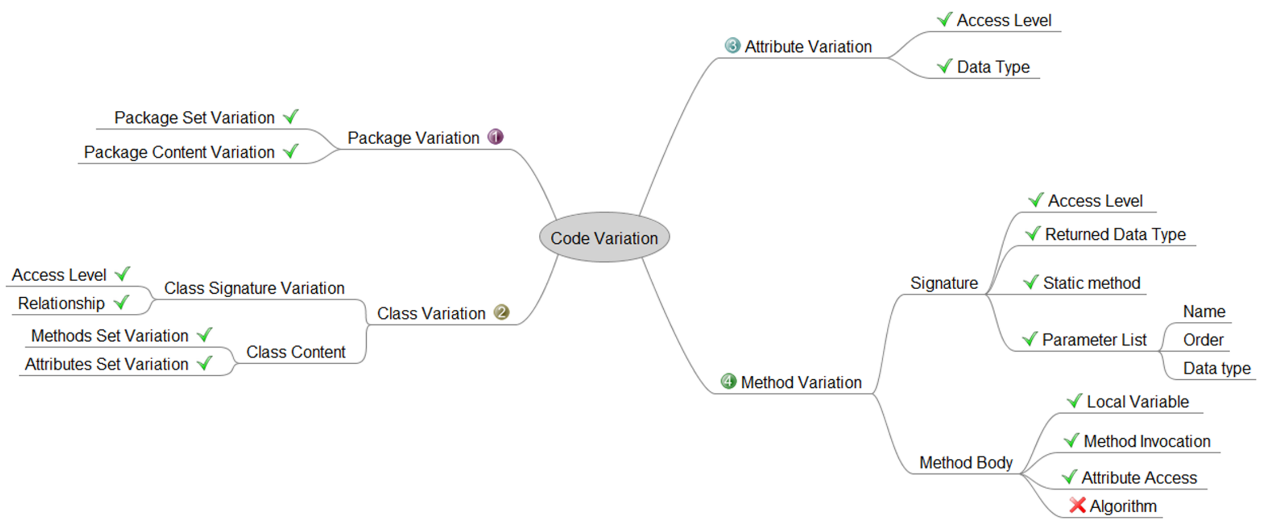


Figure 4.2 : Object-oriented source code variation.

For method body variation, in our approach we only consider local variable, method invocation and attribute access. In our work, we don't consider the *algorithms* of the method. This is not meant that the algorithms not useful for variability extraction. If we had considered a product line in which the variability was mainly represented in the body of methods (*i.e.* different choices of algorithms), our approach would present much lower detection rates. In the future it will be interesting to investigate the detection of features by considering the code of the body of methods (*i.e.* algorithms).

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

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

❶ Separate the OBE set in two subsets, the common features set – also called *common block* (CB) – and the optional features set, on which the same search process (*i.e.* reduce this search space) will have to be performed. Indeed, as optional (*resp.* common) features appear in some but not all (*resp.* all) variants, they are implemented by OBEs that appear in some but not in all (*resp.* all) variants.

❷ Separate the optional feature set into small subsets that each contains OBEs shared by groups of two or more variants or OBEs that are hold uniquely by a given variant. Each of these subsets is called a *block of variation* (BV). BVs can then be considered as smaller search spaces that each corresponds to the implementation of one or more features.

❸ Identify common atomic blocks (CAB) amongst the common block based on the expected lexical similarity (or structural similarity) between the OBEs that implement a given feature. A CB is thus composed of several CABs.

❹ Identify atomic blocks of variation (ABV) inside of each BV based on the expected lexical similarity (or structural similarity) between the OBEs that implement a given feature. A BV is thus composed of several ABVs.

All the concepts we defined for mining features are illustrated in the OBE to feature mapping model (*cf.* Figure 4.3).

REVPLINE source code model. The source code model (*cf.* the left bottom part of Figure 4.3) represents the main source code elements and their inter-relations. In our work we call such source code elements as Object-oriented building elements (OBEs). In most cases, you get enough information if you consider the core types entities that build an object-oriented system. These are *package*, *class*, *attribute*, *method*, and the relationships between them, namely *inheritance*, *access* and *invocation*. An invocation represents a method calling another method and an access represents a method accessing an attribute. These abstractions are needed for reengineering tasks such as dependency analysis. While the source code model is fairly complete, it can be easily extended in order to include other language elements. The source code model

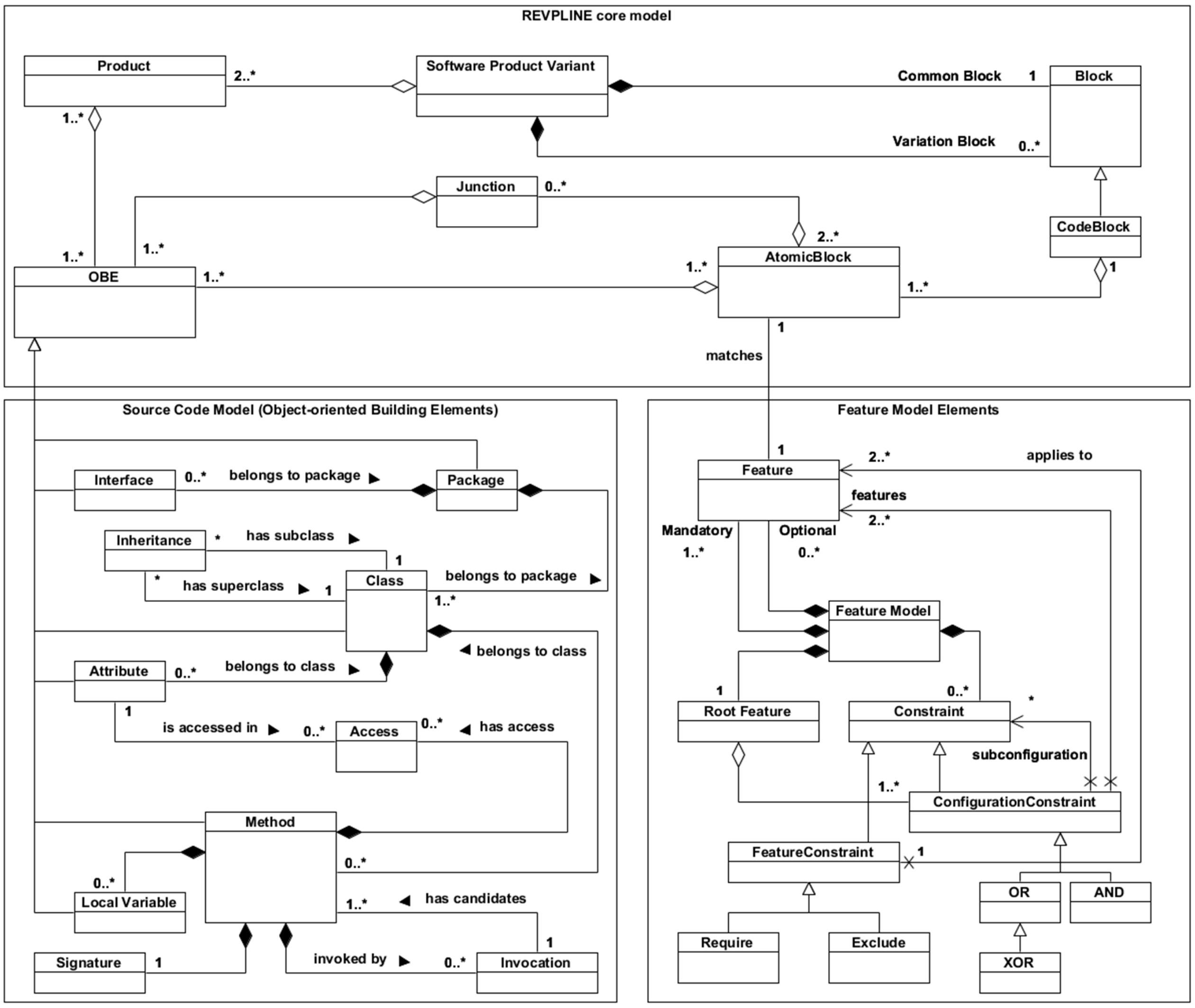


Figure 4.3 : *OBE to feature* mapping model.

does not only represent structural source code entities such as packages, classes, methods. It also represents explicitly information that is extracted from the methods' abstract syntax trees a class inherits from another class (*i.e.* inheritance), a method accesses attributes (*i.e.* access) and a method invokes other methods (*i.e.* invocation).

REVPLINE feature model. The goal of *OBE to feature* mapping model is to represent correspondences between OBE and feature model. The extracted FM must include the mined and documented functional features (optional and mandatory features) and the constraints between those features. The right bottom part of Figure 4.3 shows the REVPLINE feature model. Such model has been defined taking into account that every element will have a different graphical representation. In that Figure, a FM is represented by means of the feature model class, and a feature model can be seen as a set of features and the set of relationships among them. Features through the FM are classified into mandatory and optional features. There are two types of constraints between these features: ❶ group of features (set of features linked to the same parent such as xor-group, and-group and or-group) and ❷ cross-tree constraints (such as requires and excludes constraints). A FM must also have a root, which is denoted by means of the root feature class. Mandatory and optional features in this model represent the mined and documented features from software variants. Each atomic block corresponds to a feature. The root feature represents the name of the complete system (software family). Feature constraints (dependencies) between those features define the software configurations.

REVPLINE core model. The core model of REVPLINE approach introduces the main concepts and their inter-relations. REVPLINE core model defines the core where the other models define respectively the input and output of REVPLINE process. We introduce the main concepts of REVPLINE core model and their relations as the following: software product variants consist of 2 or more software; each software product variant consists of OBEs such as package, class, attribute, method; the OBEs of all software variants are separated into three subsets: common block (CB) (*i.e.* mandatory features set), blocks of variation (BVs) (*i.e.* optional features sets) and a set of junctions (*i.e.* features overlap); software product variants consist of only one common block and one or more blocs of variation (zero or more of junctions); the common block consists of one or more atomic block (*i.e.* mandatory feature); each block of variation consists of one or more atomic block (*i.e.* optional feature); each atomic block corresponds to one or more OBEs.

4.2.4 The Lexical Versus Structural Similarity

This section quickly introduces the lexical and structural similarity between OBEs. As a first technique, we rely on *lexical similarity* between OBEs to split the blocks of OBEs. For two OBEs the lexical similarity is based on LSI method. Whether two OBEs are lexically similar depends on the textual similarity between these OBEs. Thus lexical similarity between two OBEs is computed as a cosine of the angle between their corresponding vectors. Each pair of OBEs has similarity greater than a given threshold considered similar.

We also use a second technique, which is code dependency (*i.e.* structural similarity) between OBEs to increase the precision and recall of LSI method. When the similarity link between two OBEs were correct, then the dependency information could help locate additional correct links. Our process is based on the identification of OBEs and their relationships such as inheritance, composition, invocation relationship and so on. We use a coupling metric which measures the degree to which a class is linked to one another. Coupling is a measure of the association, whether by inheritance or by another relations, between classes in a software product [Budhkar and Gopal, 2012]. We are concerned with coupling between classes and we will consider these main dependencies [Hamdouni *et al.*, 2010]:

❶ *Inheritance coupling*: When a general class (super-class) is connected to its specialized classes (sub-classes).

❷ *Method invocation coupling*: When methods of one class use methods of another class.

❸ *Composition coupling*: When an instance of one class is referred to in another class.

❹ *Attribute access coupling*: When methods of one class use attributes of another class.

❺ *Combined coupling*: It is the union of the other couplings (*i.e.* two or more couplings) between two classes.

In this chapter we rely on structural similarity between OBEs to refine the splitting of blocks. For two OBEs the structural similarity is based on coupling. Whether two OBEs are structurally similar depends on the degree of coupling between these OBEs. Thus structural similarity between two OBEs is computed based on the coupling measures.

4.2.5 An Illustrative Example: Drawing Shapes Software Variants

As an illustrative example, we consider five drawing shapes software variants. These software product variants are developed by our team. We should mention that these software variants are developed by *copy-paste-modify* technique based on the initial release. The purpose of this case study is to illustrate our approach. This software allows a user to draw seven different kinds of shapes. Drawing shapes software variants represent a small case study (*e.g.* version 5 consists of 8 packages, 25 classes and 600 lines of code)[1].

Table 4.1 : The features of drawing shapes software product variants.

	Draw_line	Insert_image	Draw_arc	Insert_text	Draw_oval	Draw_rectangle	Draw_ThreeDRectangle	Copy	Paste
Drawing Shapes Software 1	✗	✗							
Drawing Shapes Software 2	✗	✗	✗	✗					
Drawing Shapes Software 3	✗	✗			✗				
Drawing Shapes Software 4	✗	✗				✗			
Drawing Shapes Software 5	✗	✗	✗	✗	✗	✗	✗		
Drawing Shapes Software 6	✗	✗	✗	✗	✗	✗	✗	✗	✗

Product comparison matrix
(✗ feature is in the product)

Drawing shapes software 1 supports core drawing shapes features: *draw_line* and *insert_image*. *Drawing shapes software 2* supports *draw_arc* and *insert_text* features, together with the core ones. *Drawing shapes software 3* has the core drawing shapes features and a new *draw_oval* feature. *Drawing shapes software 4* has the core drawing shapes features and a new *draw_rectangle* feature. *Drawing shapes software 5* supports *all optional features*, together with the core ones. *Drawing shapes software 6* supports *copy* and *paste* features (in addition to previous optional features), together with the core ones (*cf.* Table 4.1).

The FM in Figure 4.4 shows the FM of the drawing shapes software variants as manually designed by our team. This FM represents a basic FM without cross-tree constraints. Abstract features are not concrete features in the source code. They correspond to a group of features or to the root feature in the FM. In this example, the eventually mined features will be presented to better explain some parts of our work. However, we only use the source code of software variants as input of the mining process and thus we do not know features in advance.

[1] https://code.google.com/p/svariants/

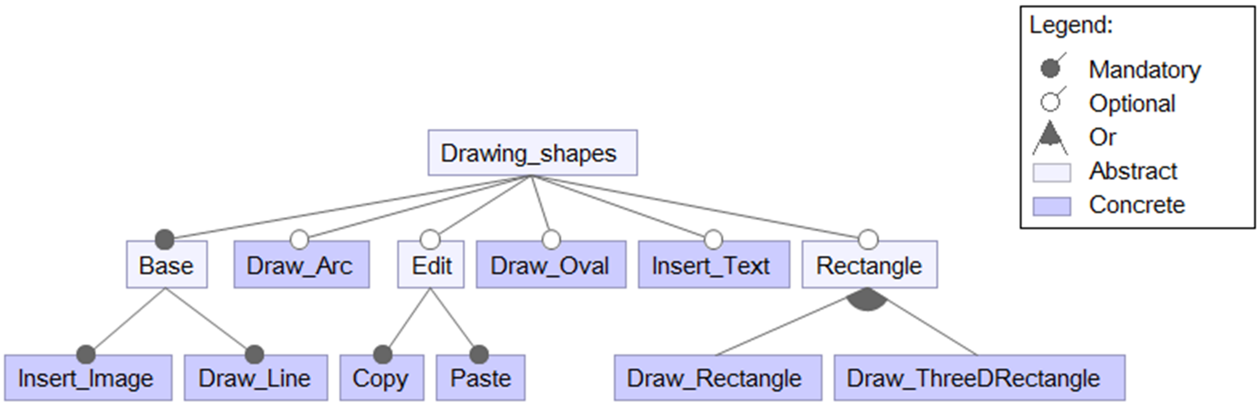


Figure 4.4 : Drawing shapes software variants feature model.

4.3 The Feature Mining Process into Details

Feature location[2] techniques aim at locating software artifacts that implement specific program functionality, *a.k.a.* a feature. In this section, we present the features mining process from the OO source code of a collection of software variants. Our approach is based on the identification of the implementation of product variant features. OBEs constitute the initial search space. We reduce this search space by first separating common and variable OBEs and, secondly, dividing the set of variable OBEs in subgroups. To reduce the search space, we rely on the commonality and variability across software variants source code. We further use lexical and structural similarity to define a similarity measure that enables to identify subgroups of OBEs that characterize the implementation of each possible feature.

The mapping model between OBEs and features defines associations between these features and the corresponding OBEs. To determine instances of this model, we describe our feature mining process. This process takes the variants' source code as its input. ❶ The first step is extracting OBEs by static analysis of the source code. ❷ The second step of this process aims at identifying BVs and the CB based on FCA (*i.e.* reduce search space) (*cf.* Section 4.3.1). ❸ The third step explores the AOC-poset of BVs to define an order to search for atomic blocks of variation (*cf.* Section 4.3.2.1). ❹ In the fourth step, we rely on lexical (LSI method) and structural (code dependency) similarity to determine the similarity between OBEs (*cf.* Section 4.3.2.2). This similarity measure is used to identify atomic blocks based on OBE clusters in the last step ❺ (*cf.* Section 4.3.2.3). Figure 4.5 shows our feature mining process.

[2]Terms for the same concept are "feature mining", "feature location" and "feature identification".

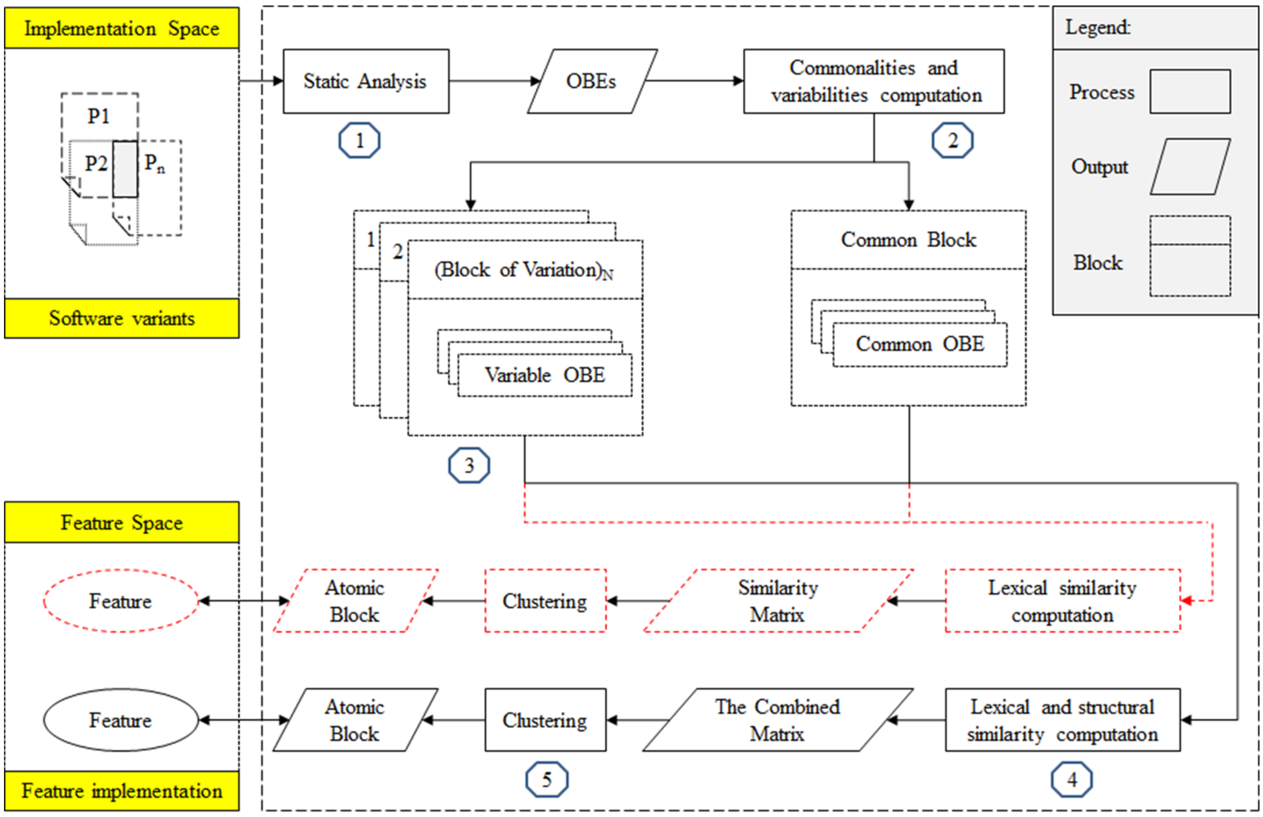


Figure 4.5 : The feature mining process.

4.3.1 Identifying the Common Block and Blocks of Variation

The first step of our feature mining process is the identification of the common OBE block and of OBE blocks of variation. The role of these blocks is to be sub-search spaces for mining sets of OBEs that implement features. The technique used to identify the CB and BVs relies on FCA. First, a formal context, where *objects* are product variants and *attributes* are OBEs (*cf.* Table 4.2), is defined. The corresponding AOC-poset is then calculated.

In the AOC-poset, the intent of each concept represents OBEs common to two or more products. As concepts of AOC-posets are ordered, the intent of the most general (*i.e.* top) concept gathers OBEs that are common to all products. They constitute the CB. The intents of all remaining concepts are BVs. They gather sets of OBEs common to a subset of products and correspond to the implementation of one or more features. The extent of each of these concepts is the set of products having these OBEs in common (*cf.* Figure 4.6).

Table 4.2 : A formal context describing drawing shapes software variants by their features.

	Class (PaintJPanel_Drawing.Shapes.Core)	Class (DrawingShapes_Drawing.Shapes.Core)	Class (MyShape_Drawing.Shapes.Core)	Class (LineSettings_Drawing.Shapes.Line)	Class (LinePosition_Drawing.Shapes.Line)	Class (MyLine_Drawing.Shapes.Line)	Class (ImagePath_Drawing.Shapes.Image)	Class (MyImage_Drawing.Shapes.Image)	Class (ImagePostion_Drawing.Shapes.Image)	Class (ArcSettings_Drawing.Shapes.Arc)	Class (Arc_Drawing.Shapes.Arc)	Class (ArcAngle_Drawing.Shapes.Arc)	Class (MyTextShape_Drawing.Shapes.Text)	Class (Text_Drawing.Shapes.Text)	Class (TextInfo_Drawing.Shapes.Text)	Class (MyText_Drawing.Shapes.Text)	Class (OvalSettings_Drawing.Shapes.Oval)	Class (Oval_Drawing.Shapes.Oval)	Class (MyOval_Drawing.Shapes.Oval)	Class (RectangleSettings_Drawing.Shapes.Rectangle)	Class (MyRectangle_Drawing.Shapes.Rectangle)	Class (Rectangle_Drawing.Shapes.Rectangle)	Class (ThreeDRectangleSettings_Drawing.Shapes.ThreeDRectangle)	Class (ThreeDRectangle_Drawing.Shapes.ThreeDRectangle)	Class (MyThreeDRectangle_Drawing.Shapes.ThreeDRectangle)	Package (Editor.Copy)	Class (CopyText_Editor.Copy)	Method (CopySettings_CopyText)	Package (Editor.Paste)	Class (PasteText_Editor.Paste)	Method (PasteSettings_PasteText)
Software_1	✗	✗	✗	✗	✗	✗	✗	✗	✗																						
Software_2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗															
Software_3	✗	✗	✗	✗	✗	✗	✗	✗	✗								✗	✗	✗												
Software_4	✗	✗	✗	✗	✗	✗	✗	✗	✗											✗	✗	✗									
Software_5	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗						
Software_6	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

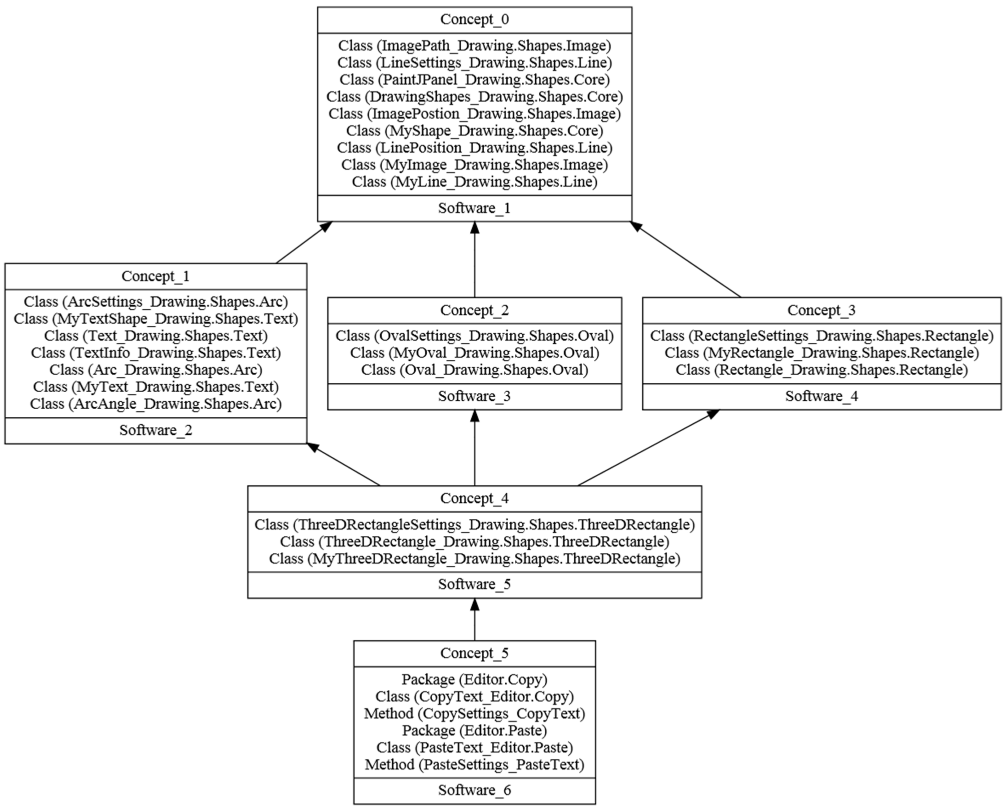

Concept_0
Class (ImagePath_Drawing.Shapes.Image)
Class (LineSettings_Drawing.Shapes.Line)
Class (PaintJPanel_Drawing.Shapes.Core)
Class (DrawingShapes_Drawing.Shapes.Core)
Class (ImagePostion_Drawing.Shapes.Image)
Class (MyShape_Drawing.Shapes.Core)
Class (LinePosition_Drawing.Shapes.Line)
Class (MyImage_Drawing.Shapes.Image)
Class (MyLine_Drawing.Shapes.Line)
Software_1
Concept_1
Class (ArcSettings_Drawing.Shapes.Arc)
Class (MyTextShape_Drawing.Shapes.Text)
Class (Text_Drawing.Shapes.Text)
Class (TextInfo_Drawing.Shapes.Text)
Class (Arc_Drawing.Shapes.Arc)
Class (MyText_Drawing.Shapes.Text)
Class (ArcAngle_Drawing.Shapes.Arc)
Software_2
Concept_2
Class (OvalSettings_Drawing.Shapes.Oval)
Class (MyOval_Drawing.Shapes.Oval)
Class (Oval_Drawing.Shapes.Oval)
Software_3
Concept_3
Class (RectangleSettings_Drawing.Shapes.Rectangle)
Class (MyRectangle_Drawing.Shapes.Rectangle)
Class (Rectangle_Drawing.Shapes.Rectangle)
Software_4
Concept_4
Class (ThreeDRectangleSettings_Drawing.Shapes.ThreeDRectangle)
Class (ThreeDRectangle_Drawing.Shapes.ThreeDRectangle)
Class (MyThreeDRectangle_Drawing.Shapes.ThreeDRectangle)
Software_5
Concept_5
Package (Editor.Copy)
Class (CopyText_Editor.Copy)
Method (CopySettings_CopyText)
Package (Editor.Paste)
Class (PasteText_Editor.Paste)
Method (PasteSettings_PasteText)
Software_6

Figure 4.6 : The AOC-poset for the formal context of Table 4.2.

4.3.2 Identifying Atomic Blocks

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

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

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

4.3.2.2 Measuring OBEs' Similarity

In this section, we present two strategies to splitting the CB and BVs into atomic blocks of OBEs. We rely on lexical similarity when variability is represented at different levels of object-oriented source code. In the context that the variability across software product variants is represented on the package and class levels we rely on lexical and structural similarity to identify feature implementations. To select the best strategy from the previous strategies, we can delegate to the expert for the target software variants. In case there is a lack of knowledge about existing software variants source code we prefer to test both strategies. In our work, we don't use code dependency between OBEs alone. We tested code dependency alone on several cases and the obtained results are not promising. We think that the obtained semantics by applying code dependency alone is low. In our approach, we use code dependency to get more links (*i.e.* similarity between OBEs) to improve the precision and recall of LSI method. In this section, we present firstly measuring of OBEs' similarity based on LSI. Then, we present measuring of OBEs' similarity based on LSI and the code dependency.

1.3.2.2.1 Measuring OBEs' Similarity Based on LSI

Here, we present measuring of OBEs' similarity based on LSI (*i.e.* lexical similarity). OBEs of BVs or of the CB respectively characterize the implementation of optional and mandatory features. We base the identification of subsets of OBEs, which each constitutes a feature, on the measurement of lexical similarity between these OBEs. We rely on the fact that OBEs involved in implementing a functional feature are lexically closer to one another than to the rest of OBEs. To compute similarity between two OBEs in the CB and BVs, we proceed in three steps: building the LSI corpus, building the term-document matrix and the term-query matrix for each BV and for the CB, building the cosine similarity matrix and, at last, transforming the cosine similarity matrix into formal context.

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

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

Table 4.3 : The term-document and the term-query matrices of ($Concept$_5) in Figure 4.6.

	Class (CopyText_Editor.Copy)	Class (PasteText_Editor.Paste)	Method (CopySettings_CopyText)	Method (PasteSettings_PasteText)	Package (Copy)	Package (Paste)
copy	2	0	2	0	1	0
paste	0	2	0	2	0	1
settings	0	0	1	1	0	0
text	1	1	1	1	0	0

The term-document matrix

	Class (CopyText_Editor.Copy)	Class (PasteText_Editor.Paste)	Method (CopySettings_CopyText)	Method (PasteSettings_PasteText)	Package (Copy)	Package (Paste)
copy	2	0	2	0	1	0
paste	0	2	0	2	0	1
settings	0	0	1	1	0	0
text	1	1	1	1	0	0

The term-query matrix

The most important parameter of LSI is the number of chosen term-topics. A term-topic is a collection of terms that co-occur frequently in the documents of the corpus. We need enough term-topics to capture real term relations. In our work we cannot use a fixed number of topics for LSI because we have blocks of variation (*i.e.* partitions) with different sizes. The number of term-topics (# term-topics) is equal to "$K * N$", where K is a variable, its value depends on the size of each BV and N is the number of columns of the term-document matrix that is generated by LSI [Al-Msie'deen *et al.*, 2013b]. For instance, in the BV of ($Concept$_5) in Figure 4.6, the K value for this BV is equal to 0.34 and the number of columns of the term-document matrix is equal to 6 (*i.e.* the number of topics in this block is equal to 2; *i.e.* $0.34 \times 6 = 2$).

❸ Building the similarity matrix. Similarity between OBEs in each BV or in the CB is described by a cosine similarity matrix whose columns and rows both represent vectors of OBEs: documents as columns and queries as rows. Similarity is computed as a cosine similarity. The similarity matrix corresponding to the BV of *Concept_5* from Figure 4.6 is presented in Table 4.4.

In our work, we represent the similarity values between the OBEs. The results are represented as a directed graph. OBEs are represented as vertices and the similarity links as edges. The degree of similarity appears along the edges of the graph (*cf.* Figure 4.7).

❹ Transforming the cosine similarity matrix into formal context. To transform the (numerical) similarity matrices of previous step into (binary) formal contexts, we use a threshold. 0.70 is

Table 4.4 : The similarity matrix of ($Concept\_5$) in Figure 4.6.

	Class (CopyText_Editor.Copy)	Class (PasteText_Editor.Paste)	Method (CopySettings_CopyText)	Method (PasteSettings_PasteText)	Package (Editor.Copy)	Package (Editor.Paste)
Class (CopyText_Editor.Copy)	1.0	0.035384	1.0	0.031342	0.999467	0.001207
Class (PasteText_Editor.Paste)	0.035384	1.0	0.035384	0.999991	0.002748	0.999415
Method (CopySettings_CopyText)	1.0	0.035384	1.0	0.031342	0.999467	0.001207
Method (PasteSettings_PasteText)	0.031342	0.999991	0.031342	0.999999	-0.001295	0.999545
Package (Editor.Copy)	0.999467	0.002748	0.999467	-0.001295	1.0	-0.031431
Package (Editor.Paste)	0.001207	0.999415	0.001207	0.999545	-0.031431	1.0

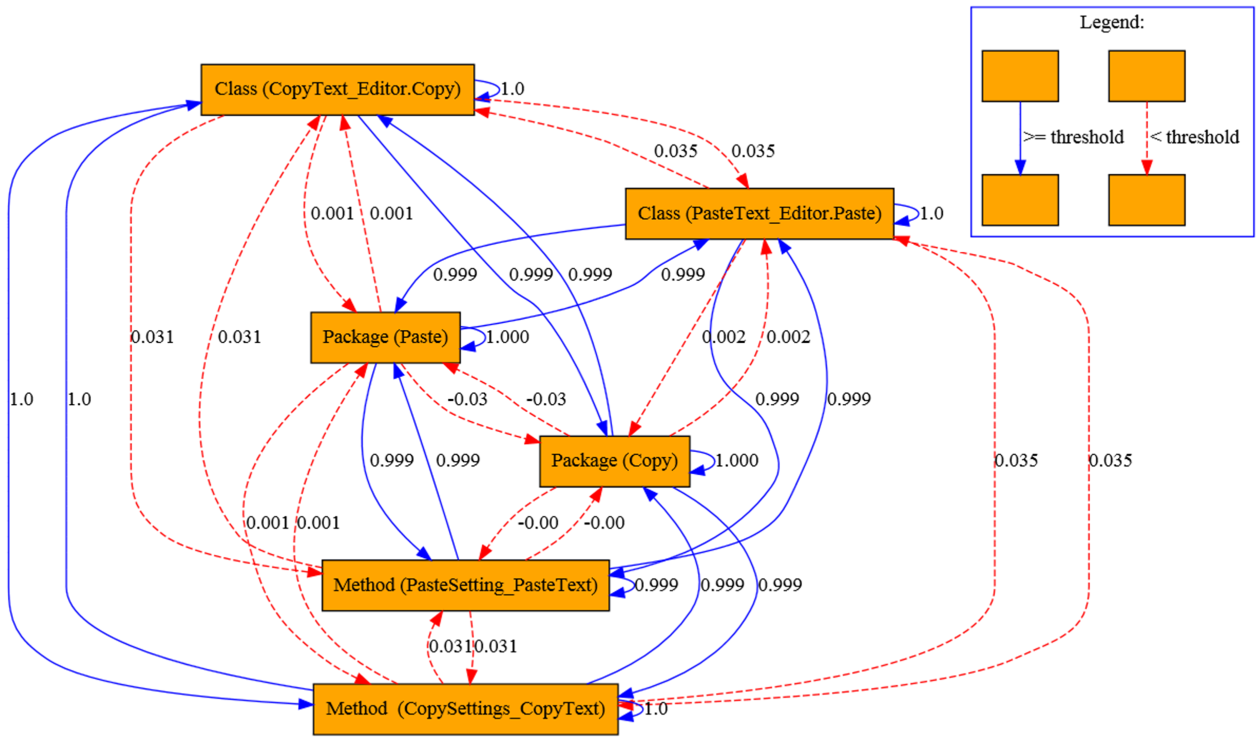

Legend:
>= threshold
< threshold
Class (CopyText_Editor.Copy)
Class (PasteText_Editor.Paste)
Package (Paste)
Package (Copy)
Method (PasteSetting_PasteText)
Method (CopySettings_CopyText)
1.0
0.035
0.001
0.999
0.002
0.031
-0.03
1.000
-0.00
0.0310.031

Figure 4.7 : The lexical similarity between OBEs of ($Concept\_5$) in Figure 4.6 as a directed graph.

the chosen threshold value (a widely used threshold for cosine similarity [Marcus and Maletic, 2003]) meaning that only pairs of OBEs having a calculated similarity greater than or equal to 0.70 are considered similar. Table 4.5 shows the formal context obtained by transforming the similarity matrix corresponding to the BV of *Concept_5* from Figure 4.6. As an example, in the formal context of this table, the OBE *"Method (PasteSetting_PasteText)"* is linked to the OBE *"Class (PasteText_Editor.Paste)"* because their similarity equals 0.99, which is greater than the threshold. However, the OBE *"Method (CopySettings_CopyText)"* and the OBE *"Class (PasteText_Editor.Paste)"* are not linked because their similarity equals 0.035, which is less than the threshold.

Table 4.5 : Formal context of ($Concept\_5$) in Figure 4.6.

	Class (CopyText_Editor.Copy)	Class (PasteText_Editor.Paste)	Method (CopySettings_CopyText)	Method (PasteSettings_PasteText)	Package (Editor.Copy)	Package (Editor.Paste)
Class (CopyText_Editor.Copy)	✗		✗		✗	
Class (PasteText_Editor.Paste)		✗		✗		✗
Method (CopySettings_CopyText)	✗		✗		✗	
Method (PasteSettings_PasteText)		✗		✗		✗
Package (Editor.Copy)	✗		✗		✗	
Package (Editor.Paste)		✗		✗		✗

1.3.2.2.2 Measuring OBEs' Similarity Based on LSI and Code Dependency

Here, we present measuring of OBEs' similarity based on LSI and code dependency (*i.e.* structural similarity). This approach aims to enhance the use of FCA and LSI, with the use of structural dependencies between OBEs. We consider only variants whose variability is expressed by the existence or not of some packages and classes. Based on our previous experience on the case studies, we found that a feature was at most implemented at the package or class level (*e.g.* ArgoUML-SPL).

To compute lexical and structural similarity between two OBEs in the CB and BVs, we proceed in three steps: measuring OBEs' similarity based on structural dependency, measuring OBEs' similarity based on LSI and, at last, combining lexical and structural similarity of OBEs.

❶ Measuring OBEs' similarity based on structural dependency. Structural similarity is used to capture and represent the dependencies between classes of a block. We use a Dependency Structure Matrix (DSM), which is a square matrix in which the classes being analyzed correspond to the rows and columns. An entry in the matrix indicates that the class on the corresponding column depends on the class on the corresponding row. Dependencies on the diagonal, from upper left to lower right, are not of interest because they would only indicate that an item depends on itself. In DSM, a character '✗' means that a dependency exists (*cf.* Table 4.6). We used a class DSM to represent the dependency among classes in the common block and inside each block of variation. We use the structural relations that are introduced in Section 4.2.4. For example, in the dependency structure matrix (*cf.* Table 4.6), the OBE "Class (ArcSettings_Drawing.Shapes.Arc)" is linked to the OBE "Class (ArcAngle_Drawing.Shapes.Arc)" because there is a structural link between these two classes (*i.e.* inheritance coupling). However, the OBE "Class (Text_Drawing.Shapes.Text)" and the OBE "Class (ArcAngle_Drawing.Shapes.Arc)" are not linked because there is no structural link between these two classes.

❷ Measuring OBEs' Similarity Based on LSI. In order to apply the LSI, we follow the same steps that are mentioned previously. To create a document for each class, we must consider all useful information (*i.e.* identifier names) that describes the class (*i.e.* package name, class name,

Table 4.6 : The dependency structure matrix of ($Concept\_1$) in Figure 4.6.

	Class (ArcSettings_Drawing.Shapes.Arc)	Class (Arc_Drawing.Shapes.Arc)	Class (ArcAngle_Drawing.Shapes.Arc)	Class (MyTextShape_Drawing.Shapes.Text)	Class (Text_Drawing.Shapes.Text)	Class (TextInfo_Drawing.Shapes.Text)	Class (MyText_Drawing.Shapes.Text)
Class (ArcSettings_Drawing.Shapes.Arc)		✗	✗				
Class (Arc_Drawing.Shapes.Arc)	✗		✗				
Class (ArcAngle_Drawing.Shapes.Arc)	✗	✗					
Class (MyTextShape_Drawing.Shapes.Text)					✗		
Class (Text_Drawing.Shapes.Text)				✗		✗	✗
Class (TextInfo_Drawing.Shapes.Text)					✗		
Class (MyText_Drawing.Shapes.Text)					✗		

attributes names, methods names and method body elements names, *e.g.* parameter name, local variable name, method invocation name, attribute access name). For example, the contents of *ArcSettings* class are ArcSettings, ArcX, ArcY, drawArc, ArcPostion, getArcX, setArcX, getArcY, setArcY, getArcPostion, setArcPostion, arcPostion and ArcAngle. All identifier names are stored in a single file for each class. To compute similarity between two OBEs in the CB and BVs and produce *Lexical Similarity Matrix* (LSM), we rely on the same steps mentioned previously in the LSI method (*i.e.* ❶ building the cosine similarity matrix and ❷ transform cosine similarity matrix into the lexical similarity matrix).

For instance, in the cosine similarity matrix (*cf.* Table 4.7), the OBE "Class (ArcSettings_Drawing.Shapes.Arc)" is similar to the OBE "Class (ArcAngle_Drawing.Shapes.Arc)" because their similarity value is 0.97, which is greater than the threshold. However, the OBE "Class (Text_Drawing.Shapes.Text)" and the OBE "Class (ArcSettings_Drawing.Shapes.Arc)" are not linked because their similarity is 0.12, thus less than the threshold. In this example, the K value for this BV (*i.e.* Concept_1) is equal to 0.30 (*i.e.* the number of topics in this block is equal to 2; *i.e.* 0.30 × 7 = 2).

Lexical similarity matrix is a square matrix (*cf.* Table 4.8) where each entry $c_{i,j}$ represents a lexical similarity between class i and class j higher than a chosen threshold (here 0.70). The diagonal entries ($c_{i,i}$) always have value '✗' to indicate that a class is similar to itself. We used class LSM to represent the lexical similarity between classes in the CB and for each of the BVs. Table 4.8 shows the formal context (*i.e.* LSM) obtained by transforming the similarity matrix corresponding to the BV of *Concept_1* from Figure 4.6.

❸ Measuring OBEs' Similarity Based on Lexical and Structural Similarity. To combine both lexical and structural similarity between OBEs in the common block or blocks of variation we introduce what we call a combined matrix. A combined matrix (CM) is a square matrix which integrates the previous two matrices. In other words, this matrix represents both DSM and LSM between a set of classes. The CM is an adjacency matrix where a cell represents a link between two classes based on the structural or lexical similarity (*cf.* Table 4.9). All possible links between OBEs are considered in this matrix. The combined matrix of Table 4.9 also constitutes the formal

Table 4.7 : The similarity matrix of ($Concept\_1$) in Figure 4.6.

	Class (Arc_Drawing.Shapes.Arc)	Class (ArcAngle_Drawing.Shapes.Arc)	Class (ArcSettings_Drawing.Shapes.Arc)	Class (MyTextShape_Drawing.Shapes.Text)	Class (MyText_Drawing.Shapes.Text)	Class (Text_Drawing.Shapes.Text)	Class (TextInfo_Drawing.Shapes.Text)
Class (Arc_Drawing.Shapes.Arc)	0.999999	0.996185	0.992939	0.069353	0.049844	0.009677	0.062624
Class (ArcAngle_Drawing.Shapes.Arc)	0.996185	0.999999	0.978799	-0.017967	-0.037500	-0.077621	-0.024709
Class (ArcSettings_Drawing.Shapes.Arc)	0.992939	0.978799	1.0	0.187202	0.167972	0.128227	0.180573
Class (MyTextShape_Drawing.Shapes.Text)	0.069353	-0.017967	0.187202	1.0	0.999809	0.018216	0.999977
Class (MyText_Drawing.Shapes.Text)	0.049847	-0.037500	0.167972	0.999809	1.0	0.012192	0.999918
Class (Text_Drawing.Shapes.Text)	0.009677	-0.077621	0.128227	0.018216	0.012192	0.999999	0.034596
Class (TextInfo_Drawing.Shapes.Text)	0.062624	-0.024709	0.180573	0.999977	0.999918	0.034596	1.0

Table 4.8 : The lexical similarity matrix of ($Concept\_1$) in Figure 4.6.

	Class (Arc_Drawing.Shapes.Arc)	Class (ArcAngle_Drawing.Shapes.Arc)	Class (ArcSettings_Drawing.Shapes.Arc)	Class (MyTextShape_Drawing.Shapes.Text)	Class (MyText_Drawing.Shapes.Text)	Class (Text_Drawing.Shapes.Text)	Class (TextInfo_Drawing.Shapes.Text)
Class (Arc_Drawing.Shapes.Arc)	✗	✗	✗				
Class (ArcAngle_Drawing.Shapes.Arc)	✗	✗	✗				
Class (ArcSettings_Drawing.Shapes.Arc)	✗	✗	✗				
Class (MyTextShape_Drawing.Shapes.Text)				✗	✗		✗
Class (MyText_Drawing.Shapes.Text)				✗	✗		✗
Class (Text_Drawing.Shapes.Text)						✗	
Class (TextInfo_Drawing.Shapes.Text)				✗	✗		✗

context which is used as input for applying FCA in the next step.

We represent the lexical and structural similarity values between the OBEs. The results are represented as a directed graph. OBEs are represented as vertices and the similarity links as edges. The degree of lexical similarity appears along the edges of the graph (*cf.* Figure 4.8). This graph presents another view of the similarity between source code elements (*i.e.* OBEs). It is produced by the REVPLINE approach.

Table 4.9 : The combined matrix of ($Concept\_1$) in Figure 4.6.

	Class (Arc_Drawing.Shapes.Arc)	Class (ArcAngle_Drawing.Shapes.Arc)	Class (ArcSettings_Drawing.Shapes.Arc)	Class (MyTextShape_Drawing.Shapes.Text)	Class (MyText_Drawing.Shapes.Text)	Class (Text_Drawing.Shapes.Text)	Class (TextInfo_Drawing.Shapes.Text)
Class (Arc_Drawing.Shapes.Arc)	✗	✗	✗				
Class (ArcAngle_Drawing.Shapes.Arc)	✗	✗	✗				
Class (ArcSettings_Drawing.Shapes.Arc)	✗	✗	✗				
Class (MyTextShape_Drawing.Shapes.Text)				✗	✗	✗	✗
Class (MyText_Drawing.Shapes.Text)				✗	✗	✗	✗
Class (Text_Drawing.Shapes.Text)				✗	✗	✗	✗
Class (TextInfo_Drawing.Shapes.Text)				✗	✗	✗	✗

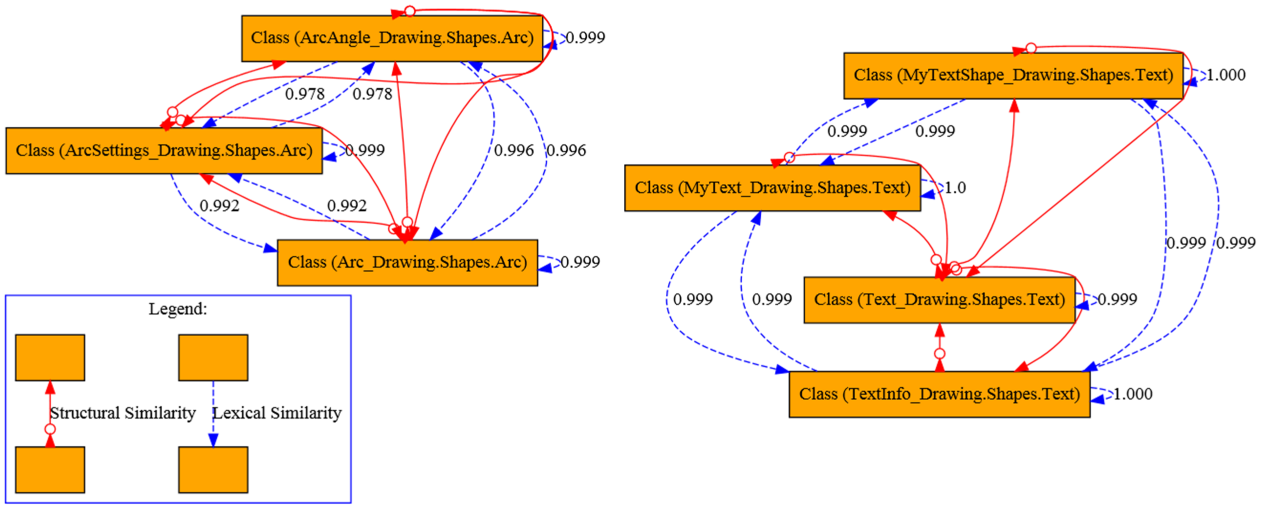

Class (ArcAngle_Drawing.Shapes.Arc)
0.999
0.978
0.978
Class (ArcSettings_Drawing.Shapes.Arc)
0.999
0.996
0.996
0.992
0.992
Class (Arc_Drawing.Shapes.Arc)
0.999
Legend:
Structural Similarity
Lexical Similarity
Class (MyTextShape_Drawing.Shapes.Text)
1.000
0.999
0.999
Class (MyText_Drawing.Shapes.Text)
1.0
0.999
0.999
Class (Text_Drawing.Shapes.Text)
0.999
0.999
0.999
Class (TextInfo_Drawing.Shapes.Text)
1.000

Figure 4.8 : The lexical and structural similarity between OBEs of ($Concept\_1$) in Figure 4.6 as a directed graph.

4.3.2.3 Identifying Atomic Blocks Using FCA

To clustering the similar source code elements of CB and BVs into atomic blocks of OBEs (*i.e.* feature implementations), we rely on FCA. Based on the formal context (*resp.* combined matrix) that is obtained in Table 4.5 (*resp.* Table 4.9); after measuring OBEs' similarity based on LSI (*resp.* LSI and code dependency); we use FCA to identify, from each block of OBEs, which elements are similar. The resulting AOC-poset is composed of concepts the extent and intent of which group similar OBEs (*i.e.* equal intent and extent).

For the drawing shapes example, the AOC-poset of Figure 4.9 shows two atomic blocks of variation[3] (that correspond to two distinct features) mined from a single block of variation (*Concept_5* from Figure 4.6) based on the formal context of Table 4.5.

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

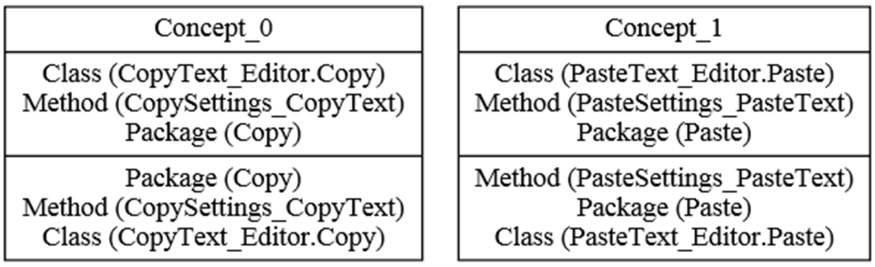

Concept_0
Class (CopyText_Editor.Copy)
Method (CopySettings_CopyText)
Package (Copy)
Package (Copy)
Method (CopySettings_CopyText)
Class (CopyText_Editor.Copy)
Concept_1
Class (PasteText_Editor.Paste)
Method (PasteSettings_PasteText)
Package (Paste)
Method (PasteSettings_PasteText)
Package (Paste)
Class (PasteText_Editor.Paste)

Figure 4.9 : Atomic Blocks Mined from ($Concept$_5) in Figure 4.6.

The AOC-poset of Figure 4.10 shows two atomic blocks of variation (that correspond to two distinct features) mined from a single block of variation (Concept_1 from Figure 4.6) based on the combined matrix of Table 4.9. The same feature mining process is used for the CB and for each of the BV. The interest of FCA for this task is to help extracting concepts which represent mutually similar OBEs [Azmeh, 2011].

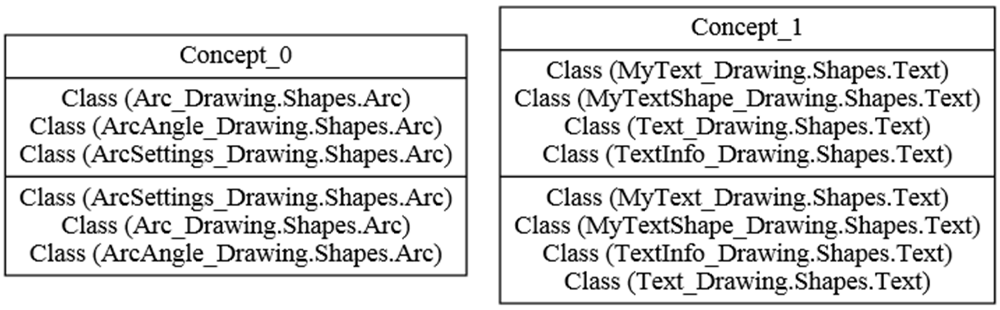

Concept_0
Class (Arc_Drawing.Shapes.Arc)
Class (ArcAngle_Drawing.Shapes.Arc)
Class (ArcSettings_Drawing.Shapes.Arc)
Class (ArcSettings_Drawing.Shapes.Arc)
Class (Arc_Drawing.Shapes.Arc)
Class (ArcAngle_Drawing.Shapes.Arc)
Concept_1
Class (MyText_Drawing.Shapes.Text)
Class (MyTextShape_Drawing.Shapes.Text)
Class (Text_Drawing.Shapes.Text)
Class (TextInfo_Drawing.Shapes.Text)
Class (MyText_Drawing.Shapes.Text)
Class (MyTextShape_Drawing.Shapes.Text)
Class (TextInfo_Drawing.Shapes.Text)
Class (Text_Drawing.Shapes.Text)

Figure 4.10 : Atomic Blocks Mined from ($Concept$_1) in Figure 4.6.

In this chapter, we mined initial FM based on the mined feature implementations[4] (*cf.* Table 4.10). For readability's sake, we manually associated feature names to atomic blocks, based on the study of the content of each block and on our knowledge on software. Of course, this does not impact the quality of our results. Based on the software configurations (*i.e.* product feature sets) we identify a basic FM which consists of optional and mandatory features with only one level of hierarchy and without cross-tree constraints (CTCs) and groups of features constraints. Figure 4.11 shows the mined FM for drawing shapes software variants.

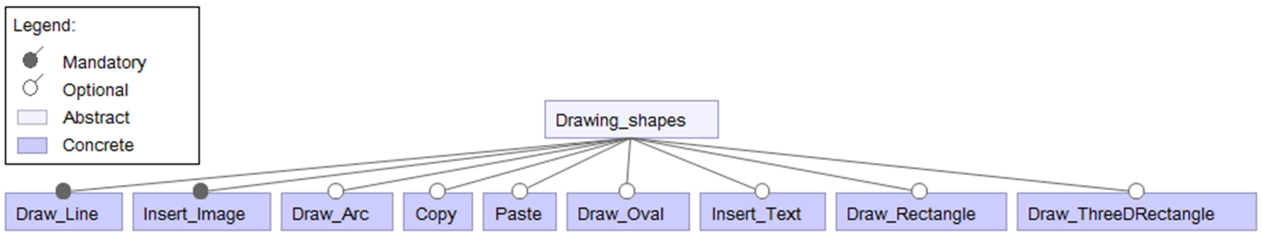


Figure 4.11 : The mined feature model of drawing shapes software variants.

[4]The source code available at `https://code.google.com/p/refm/`

Table 4.10 : Feature sets of drawing shapes software variants (*i.e.* software configurations).

	Drawing_shapes	Draw_line	Insert_image	Draw_arc	Insert_text	Draw_oval	Draw_rectangle	Draw_ThreeDRectangle	Copy	Paste
Software_1	✖	✗	✗							
Software_2	✖	✗	✗	✓	✓					
Software_3	✖	✗	✗			✓				
Software_4	✖	✗	✗				✓			
Software_5	✖	✗	✗	✓	✓	✓	✓	✓		
Software_6	✖	✗	✗	✓	✓	✓	✓	✓	✓	✓

Product-by-feature matrix
(✖ the root feature; ✗ mandatory feature; ✓ optional feature)

4.4 Threats to validity

In this chapter, we proposed an approach to identify mandatory and optional feature implementations from object-oriented source code of a set of software product variants (*i.e.* first contribution). We exploited commonality and variability across software variants at source code level to reduce the search space. In our work, we use both LSI method and code dependency to identify possible feature implementations. There are threats to validity that limit our approach in some points. In this section, we explain all threats to validity that are relevant to our study as follows:

❒ Splitting the OBEs of CB and BVs using lexical similarity is not effective in all cases. Because software developers might not use the same vocabularies to name OBEs (*e.g.* package, class, attribute, method, *etc.*) across software product variants. This means that lexical similarity may be not reliable (or should be improved with other techniques) in all cases to identify feature implementations.

❒ In our work, we cannot use a fixed number of topics for LSI method because we have blocks with different sizes. We have appointed the number of chosen term-topics (*i.e.* parameter of LSI) for each block. In our work, the expert selects the number of terms topics for each block (CB and BVs). If the expert knows the number of features for each block, he selects the same number of features as the number of terms-topics; otherwise, the expert must test some values of the number of terms-topics for each block. Selecting the appropriate number of dimensions (K) for the LSI representation is an open research question.

❒ The CB gathers all OBEs corresponding to the implementation of mandatory features in addition to source code elements that correspond to the model elements that represent the platform used to execute the product variant. In our approach, we can't distinguish model elements (the platform elements) from mandatory features. We split the CB into a set of clusters (*i.e.* atomic blocks) based on the lexical and/or structural similarity between its OBEs.

❒ For the combined approach (*i.e.* LSI method and code dependency), we only investigate product variants in which the variability is represented in the packages or classes without considering software variants where the variability is represented at method or method body levels.

❒ In current work, we consider BV that contains OBEs shared between two or more feature implementations as a feature. In fact, the junction does not consider as a feature it just a collection of OBEs common to two or more feature implementations. As perspective of this work, we plan to distinguish between feature implementations and junctions (*i.e.* feature overlaps).

❒ In this chapter, we manually assign feature names to feature implementations (*i.e.* atomic blocks), based on the study of the content of each block and our knowledge about software variants.

❒ Our approach assumes that commonality and variability across source code of software variants can be determined statically, such as product variants of ArgoUML-SPL used in our evaluation (*cf.* Chapter 7). However, there exist systems that only behave differently depending on runtime parameters. For such systems, we need to extend our approach with dynamic analysis techniques.

❒ In this chapter, we extract basic FM based on the mined feature implementations. The mined FM consists of mandatory and optional features. This FM can be only considered as a starting point. This dissertation aims to extract FM with all basic elements such as dependency between features (*i.e.* requires and excludes constraints) and group of features constraints (*i.e.* xor-group, and-group and or-group).

❒ There is a limitation using FCA as clustering technique. FCA deals with binary formal context (1, 0). This affects the quality of the result, since a similarity value 0.99 (*resp.* 0.69) is treated as a similarity value 0.70 (*resp.* 0).

4.5 Conclusion

This chapter presents the heart of this dissertation (a feature location approach called REVPLINE). We began by the philosophy of the approach where we have defined the core concepts and hypotheses of the proposed approach. Our approach accepts as input only source code of software variants. We do not make any assumptions regarding how product variants are generated and managed. REVPLINE approach is based on several techniques (FCA, LSI and code dependency). We have implemented our approach and evaluated its produced results on several case studies (small, medium and large systems). The results of this evaluation showed that most of the features were identified (*cf.* Chapter 7). REVPLINE approach considers both lexical and structural similarity to identify features.

Thus, we believe that REVPLINE offers a reasonable solution for "re-engineering software variants into SPLs". In this chapter, we present REVPLINE as an approach for feature location in a software family as we have tried to show throughout the chapter. However, remember the points that make the REVPLINE approach specific:

❶ REVPLINE accepts as input for the mining process only the source code of software product variants (*i.e.* without feature names, feature descriptions or feature model).

❷ REVPLINE mines features from different sizes of software product variants (no problem regarding scalability).

❸ REVPLINE approach exploits commonality and variability across software variants source code to reduce the search space and apply lexical and structural methods in an efficient way. REVPLINE compares multiple software variants at once[5].

❹ REVPLINE splits each block of OBEs (CB and BVs) based on the lexical or/and structural similarity between these OBEs into a set of atomic blocks (*i.e.* feature implementations). We rely on the LSI to measure the lexical similarity between OBEs and class coupling to measure structural similarity between OBEs.

❺ REVPLINE investigates software product variants in which variability is represented at different levels of OBEs (*i.e.* package, classes, attributes, methods *etc.*). REVPLINE determines the level of variation until on the smallest details such as access level of class, method or attribute.

❻ In some cases, REVPLINE can extracts all optional features from software variants source code without needing to use lexical and structural similarity; when we consider all variants such as ArgoUML-SPL (*cf.* Chapter 7). However, splitting some features from BVs is certainly much less time consuming and error-prone than browsing a large source code so that the impact of the problem is probably limited.

❼ REVPLINE mined set of feature implementations from existing software variants. Feature mining process aims to extracting information about features, commonality and variability, feature relations and feature implementation in software variants (or until SPLs). REVPLINE output is very helpful for comprehension, maintenance and reuse of software.

[5]Further studies are required to investigate the impacts of the number of software product variants on the effectiveness of our approach.

Here, we compare the REVPLINE feature location approach to the related work. Table 4.11 gives an overview of the REVPLINE approach where we present the objectives, target systems, programmed method, type of code analysis, inputs, techniques, outputs and strategies.

Table 4.11 : Summary of REVPLINE: feature location in a collection of software product variants.

Objectives						Software		Programmed method		Code analysis		Inputs					Techniques			Outputs						Languages	Case study			Strategies
Feature location	Re-engineering	Reuse	Maintenance	Comprehension	Code-to-feature traceability link	Single software	Software family	Automatic	Semi-automatic	Static	Dynamic	Package	Class	Attribute	Method	Method body	FCA	LSI	Code dependency	Junction	Common code	Variable code	Mandatory feature	Optional feature	Basic FM	Java	Small	Medium	Large	Reduce search space
✗	✗	✗	✗	✗			✗		✗	✗		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

CHAPTER 5

REVPLINE: DOCUMENTING THE MINED FEATURE IMPLEMENTATION

Get the habit of analysis – analysis will, in time, enable synthesis to become your habit of mind.

Frank Lloyd WRIGHT

Preamble

To exploit existing software variants and build a SPL, a FM of this SPL must be built as a first step. To do so, it is necessary to mine optional and mandatory features. Then, it is important to assign meaningful names for the mined feature implementations for the purpose of building a FM. In this chapter, we present a new approach to documenting the mined feature implementations from the object-oriented source code of a collection of software product variants. Section 5.1 gives an introduction of this chapter. It also presents our goal and motivation. Section 5.2 explains how use-cases can be useful for features documentation and presents the illustrative example. Section 5.3 presents the feature documentation principles. Section 5.4 gives an overview of the feature documentation process. Section 5.5 details the feature documentation process step by step. Section 5.6 presents the feature documentation process based on OBE names. Next Section 5.7 discusses threats to the validity of the feature documentation process. Finally in Section 5.8, we conclude this chapter.

5.1 Introduction

As stated previously, the main goal of this dissertation is to re-engineering software product variants into SPL. In the previous chapter of this dissertation, we have presented an approach for feature location from object-oriented source code of a collection of software product variants. In this approach we mined set of feature implementations as source code elements (OBEs). However, features in FM need to be documented. It is important to assign meaningful names and descriptions for the mined feature implementations for the purpose of building a FM. The documentation of feature implementations means giving a meaningful name and description, which describe the aim of this feature and its role in the FM. This chapter proposes an approach for building such documentation.

To reach our goal and documenting the mined feature implementations, we rely on existing software variants documents such as use-case diagrams. To document the mined feature implementations we use two kinds of software variants artefacts: the set of mined feature implementations (*i.e.* sets of OBEs) and use-case diagrams. In this chapter, we propose a new approach of documenting the mined features by giving names and descriptions, based on the feature implementations and use-case diagrams of software variants. The novelty of our approach is that we exploit commonality and variability across software variants, at feature implementation and use-cases levels, to apply again the LSI method in an efficient way.

The majority of existing documentation approaches are designed to extract labels, names, topics from the source code of single software [Kebir *et al.*, 2012] [Kuhn *et al.*, 2007] [Lucia *et al.*, 2012]. Some approaches are designed to identify code-to-document (use-case) traceability link in a single software system [Diaz *et al.*, 2013] [Sridhara *et al.*, 2010] [Davril *et al.*, 2013]. LSI method had positive results in addressing comprehension and maintenance tasks, such as feature location [Xue *et al.*, 2012], recovery of traceability links between source code and different software artifacts [Grechanik *et al.*, 2007] [Dit *et al.*, 2013], naming of software components [Kuhn, 2009] and labelling of software source code [Lucia *et al.*, 2012].

In the context of feature documentation, all studied works are manually assigned feature names (without any description) to the feature implementations [Yang *et al.*, 2009] [Ziadi *et al.*, 2012]. By contrast, our approach is designed to automatically assign name and description for each feature implementation in a set of software variants based on several techniques (FCA, RCA and LSI). Feature documentation is made based on the use-cases names and their description. The goal of this documentation is to reflect feature roles at the domain level. Additionally, for purposes of constructing an FM and reusing existing features in other software, each feature implementation that is presented to the human user must have a meaningful name. In addition, feature documentation is needed in order to understand existing software variants and facilitate their maintenance.

Considering commonality and variability across software variants allows us to cluster the use-cases and feature implementations into *disjoint* and *minimal* clusters based on RCA. Each cluster is disjoint from the others and consists of a minimal subset of feature implementations and the corresponding use-cases. Then, we use LSI and FCA to define a similarity measure that enables us to identify which use-cases characterize the name and description of each feature implementation.

5.2 Specify Use-case Diagrams of Software Variants with Variability

In this chapter, we document the mined feature implementations based on the use-case diagrams of software variants. We rely on the use-case names and descriptions to propose names and descriptions for the mined feature implementations. This section explains how use-cases can be useful for feature documentation. It also describes the example that illustrates the remaining sections of the chapter.

5.2.1 Exploiting Use-cases to Support Feature Documentation

A use-case diagram is a graphical representation of external functionalities of a system and which actors are involved in these functionalities. A use-case is a methodology used in system analysis to identify, explain, and organize system requirements [Dolques *et al.*, 2012]. Figure 5.1 shows the use-case diagram of the first version of mobile media software variants.

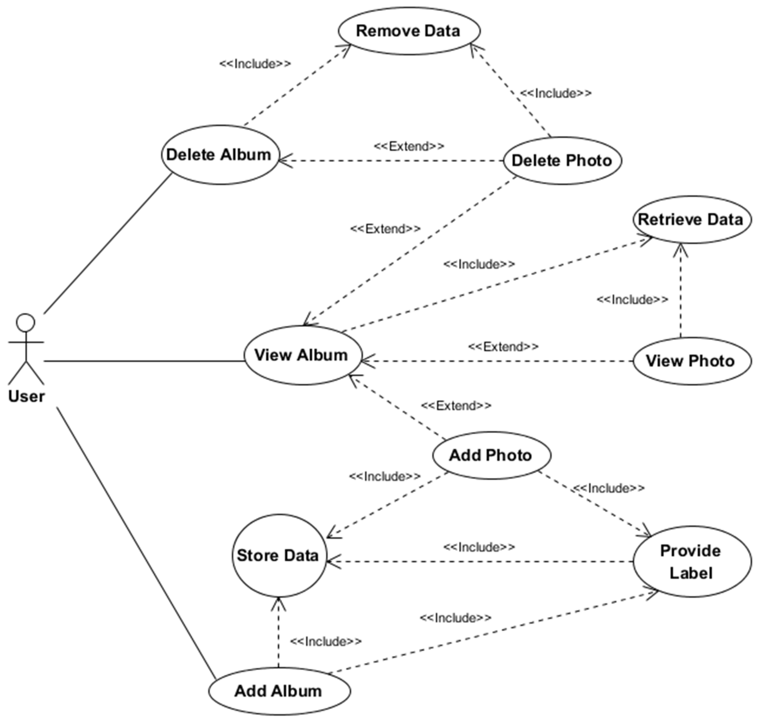


Figure 5.1 : Use-case diagram of version 1 from mobile media software [Conejero *et al.*, 2012].

Use-cases have been widely adopted since their introduction [Jacobson, 1992]. According to the UML 2.0 specification [Braganca and Machado, 2007], a use-case is the "specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system". The use-case specification gives us further detail of requirements based on legacy or software variants documentation.

Use-case diagrams of software variants can be used to represent *variability*. For example, Figure 5.1 shows the core functionalities (use-cases) for the first release of mobile media software. Figure 5.2 shows the functionalities of release 8 [Conejero *et al.*, 2012]. The first version of mobile media consists of 10 use-cases while the last version of mobile media consists of 23 use-cases.

Some non-functional requirements can be refined as non-functional use-cases. There are kinds of non-functional requirements that represent system wide qualities and are described

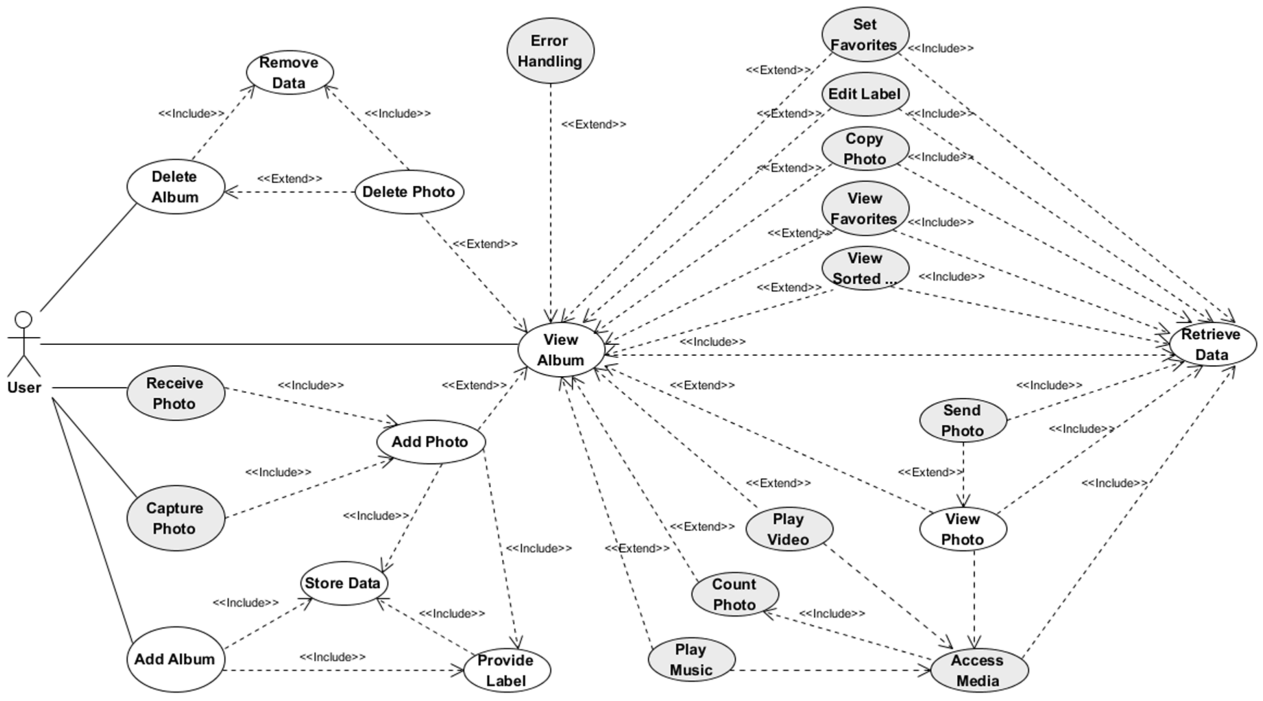


Figure 5.2 : Use-case diagram of version 8 from mobile media software[Conejero *et al.*, 2012].

simply as declarative statements during requirements [Jacobson and Ng, 2004]. Exception handling is a non-functional requirement and it is possible to specify it as a non-functional use-case [Rubira *et al.*, 2005] (*cf.* Figure 5.2).

There is an *interdependent relationship* between features and use-cases. Feature models focus on specifying the features variability by means of a graphical user-friendly and hierarchical structure. On the other hand, use-cases specify the interaction between user and system, and also the system behavior. Thus, feature models support defining the variability of each use-case and feature dependencies can be depicted in terms of the dependencies between the use-cases [Gomaa, 2004]. Use-cases are widely used to describe requirements and desired functionality of software product variants. During requirements analysis, use-case diagrams help to identify the actors and to define by means of use-cases the behavior of a system [Jacobson and Ng, 2004].

Use-cases can have relations between them. Mainly, a use-case can include (or be included by) other use-cases and can extend (or be extended by) other use-cases. Even this is not the case in our mobile media example, a use-case can also specialize another one. Such specialization is also possible between two actors. In our work, we rely on the same assumption used in the work of [Braganca and Machado, 2007] stating that each use-case represents a feature. The use-case diagrams are used to document the mined feature implementations. We rely on the use-case names and descriptions to document the mined feature implementations. We do not consider relationships between use-cases. In our work, we exploit commonality and variability across software variants at the use-cases level to document the mined feature implementations. We do not consider the relations between use-cases (*i.e.* include, extend, specialize) where it's not useful in the feature documentation process. The relations between use-case are very important for feature model identification (*i.e.* feature dependencies) such as the proposed approach in [Braganca and Machado, 2007].

5.2.2 An Illustrative Example: Mobile Tourist Guide Software Variants

As an illustrative example, we consider four software variants of Mobile Tourist Guide (MTG). Here, we should mention that MTG is a toy example to illustrate our approach. These applications allow a user to inquire about some tourist information through the mobile device. MTG_1 supports core MTG functionalities: *view map*, *place marker on a map*, *view direction*, *launch Google map* and *show street view*. MTG_2 has the core MTG functionalities and a new functionality called *download map from Google*. MTG_3 has the core MTG functionalities and a new functionality called *show satellite view*. MTG_4 supports *search for nearest attraction*, *show next attraction* and *retrieve data* functionalities, together with the core ones. Table 5.1 describes MTG software variants by their use-cases. Figure 5.3 shows the use-case diagrams of MTG software variants.

Table 5.1 : The use-cases of MTG software variants.

	View map	Place marker on a map	View direction	Launch Google map	Show street view	Download map from Google	Show satellite view	Search for nearest attraction	Show next attraction	Retrieve data
Mobile Tourist Guide 1	✘	✘	✘	✘	✘					
Mobile Tourist Guide 2	✘	✘	✘	✘	✘	✘				
Mobile Tourist Guide 3	✘	✘	✘	✘	✘		✘			
Mobile Tourist Guide 4	✘	✘	✘	✘	✘			✘	✘	✘

Product-by-use case matrix
(✘ use-case is in the product)

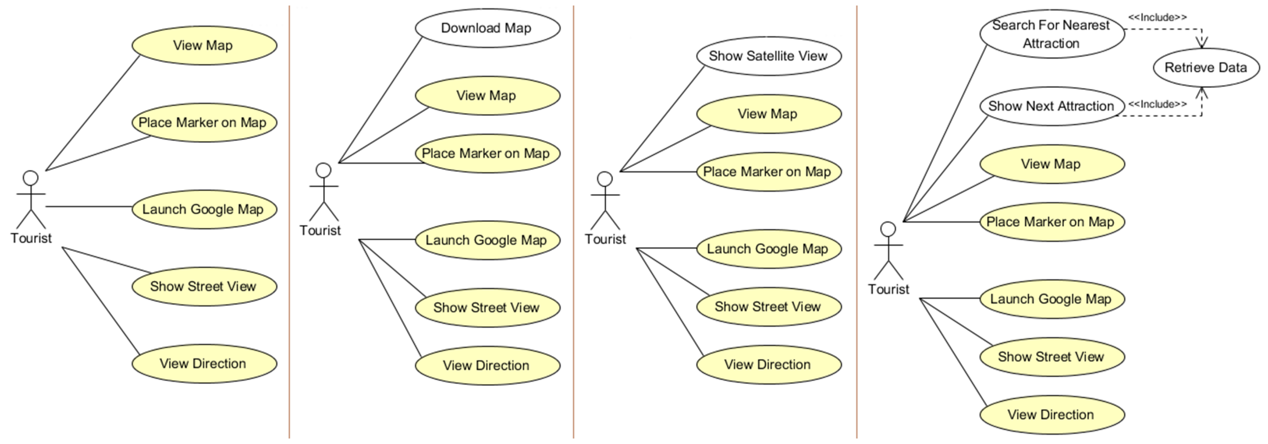


Figure 5.3 : The use-case diagrams of the four MTG software variants.

From this example, we can note that use-case diagrams of software variants show commonality and variability at use-cases level (*i.e.* functionalities). From here, the idea of feature documentation based on use-case diagrams of software variants arises. Table 5.2 shows the mined feature implementations from MTG software variants. In this example, the mined feature implementations are named using use-cases to better explain some parts of our work in this chapter, but as said before we don't know feature names in advance. However, we only use the mined feature implementations (OBEs) and use-case diagrams of software variants as input of the documentation process.

Table 5.2 : The mined feature implementations from MTG software variants.

	Feature Implementation_1: View map	Feature Implementation_2: Place marker on a map	Feature Implementation_3: View direction	Feature Implementation_4: Launch Google map	Feature Implementation_5: Show street view	Feature Implementation_6: Download map from Google	Feature Implementation_7: Show satellite view	Feature Implementation_8: Search for nearest attraction	Feature Implementation_9: Show next attraction	Feature Implementation_10: Retrieve data
Mobile Tourist Guide 1	✘	✘	✘	✘	✘					
Mobile Tourist Guide 2	✘	✘	✘	✘	✘	✘				
Mobile Tourist Guide 3	✘	✘	✘	✘	✘		✘			
Mobile Tourist Guide 4	✘	✘	✘	✘	✘			✘	✘	✘

Product-by-feature implementation matrix
(✘ feature implementation is in the product)

5.3 Principles of Feature Documentation

Documenting the mined feature implementations from the source code of software variants amounts to identify groups of use-cases that constitute feature documentation. This group of use-cases must either be present in all variants (case of a mandatory feature implementation) or in some but not all variants (case of an optional feature implementations). Thus, the initial search space for the feature documentation process is composed of all the use-cases and feature implementations in the existing product variants.

Software product variants consist of 2 or more software. These variants consist of *N* feature implementations and *M* use-cases. As the number of use-cases (*resp.* feature implementations) is high, documenting features entails to reduce this search space. Several strategies can be combined to do so:

❶ Separate the use-cases (*resp.* feature implementations) set in two subsets, the common use-cases (*resp.* feature implementations) set – also called common use-case (*resp.* feature implementations) block – and the optional use-cases (*resp.* feature implementations) set, on which the same search process will have to be performed. Indeed, as optional (*resp.* common) feature implementations appear in some but not all (*resp.* all) variants, they are documented by use-cases that appear in some but not in all (*resp.* all) variants.

❷ Separate the optional feature set into small subsets that each contains use-cases (*resp.* feature implementations) shared by groups of two or more variants or use-cases (*resp.* feature implementations) that are hold uniquely by a given variant. Each of these subsets is called a block of variation.

❸ Identify the smallest search space of use-cases and feature implementations (*i.e.* hybrid block) by linking the common use-cases block with the common feature implementations block. The same process is performed for all blocks of variation (linking use-cases with corresponding feature implementations blocks). The hybrid block can then be considered as smaller search spaces that each corresponds to the documentation of one or more feature implementations. Each hybrid block represents minimal and disjoint search spaces.

❹ Identify features documentation amongst each hybrid block based on the expected lexical similarity between the use-cases and feature implementations.

❺ Features can be documented also using the names (*i.e.* identifier names) of the OBE corresponding to atomic blocks when use-cases are *missing*.

❻ For each software family, the *mined and documented features* are stored as a matrix. We call this matrix the *product-by-feature* matrix. This matrix represents software configurations. Product-by-feature matrix (that is used in the next chapter) plays an important role to reverse engineering FM and its constraints.

All the concepts we defined for documenting features are illustrated in the extended version of *OBE to feature* mapping model (*cf.* Figure 5.4). We highlight the differences using the *dashed line* for the new concepts. The new concepts are added to the REVPLINE core model (*i.e.* process data). Each software variant has one use-case diagram, each use-case diagram consists of one or more use-cases and each use-case is characterized by its name and description. By separating the use-cases of software variants we get two subsets: the common use-case block and blocks of use-case variation. Each hybrid block consists of one use-case block and one or more atomic block (*i.e.* feature implementation). The hybrid block documents one or more of feature implementation. The documentation process relies on the use-case name and its description. Software family consists of one or more hybrid block. Features can be documented also using the names of the OBE (*i.e.* identifier names) corresponding to the atomic blocks when use-cases are missing. For each software family, the mined and documented features are stored as a product-by-feature matrix. Based on the product-by-feature matrix, we extract the feature model by using FCA as described in the next chapter.

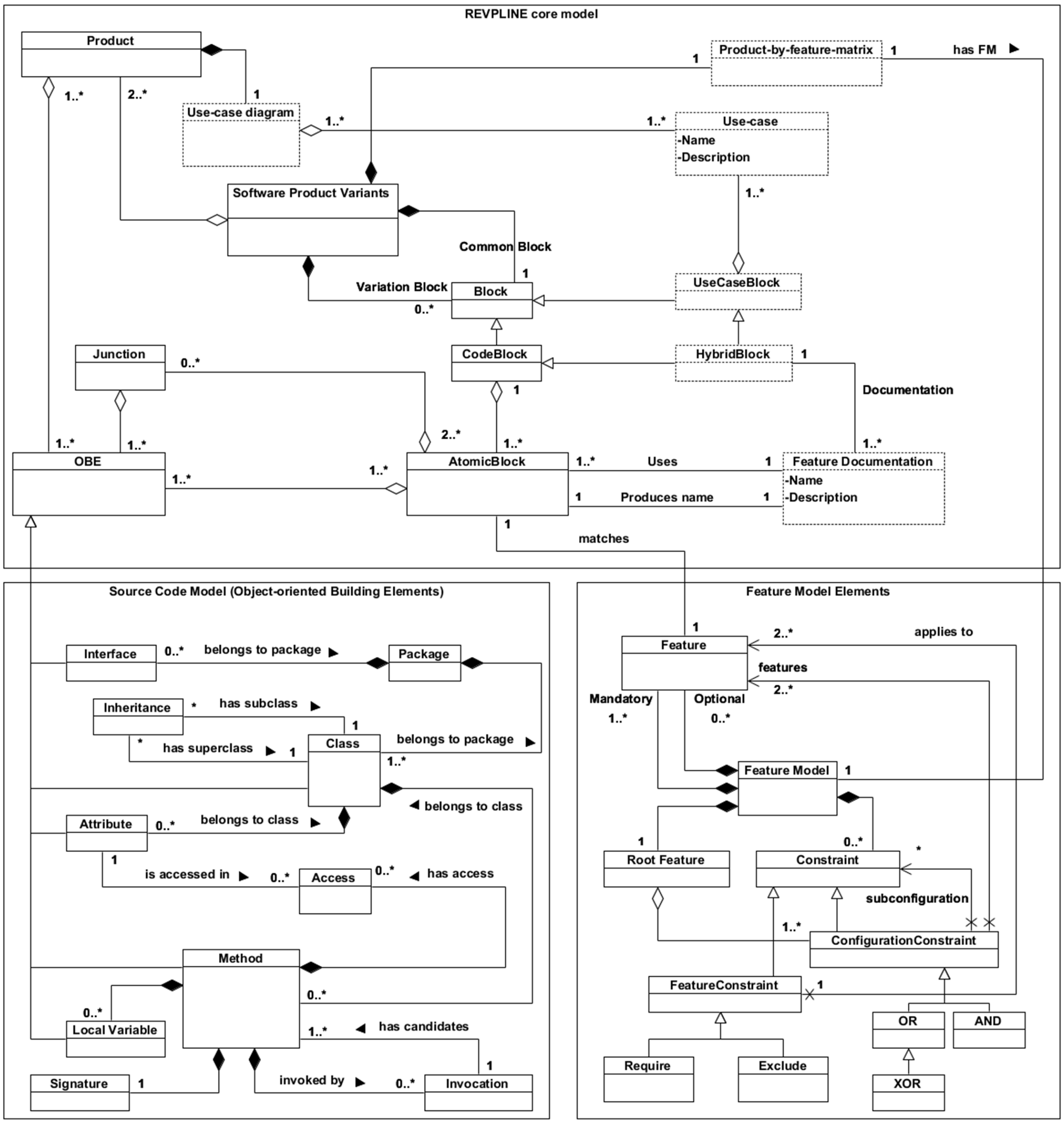


Figure 5.4 : The extended version of *OBE to feature mapping model.*

5.4 Feature Documentation Overview

Our goal is to document the mined feature implementations from a collection of software variants. Based on existing use-case diagrams of software variants, we document the mined feature implementations by combining both use-cases and feature implementations. So we are exploiting the available use-cases and source code relevant to software variants to document their features.

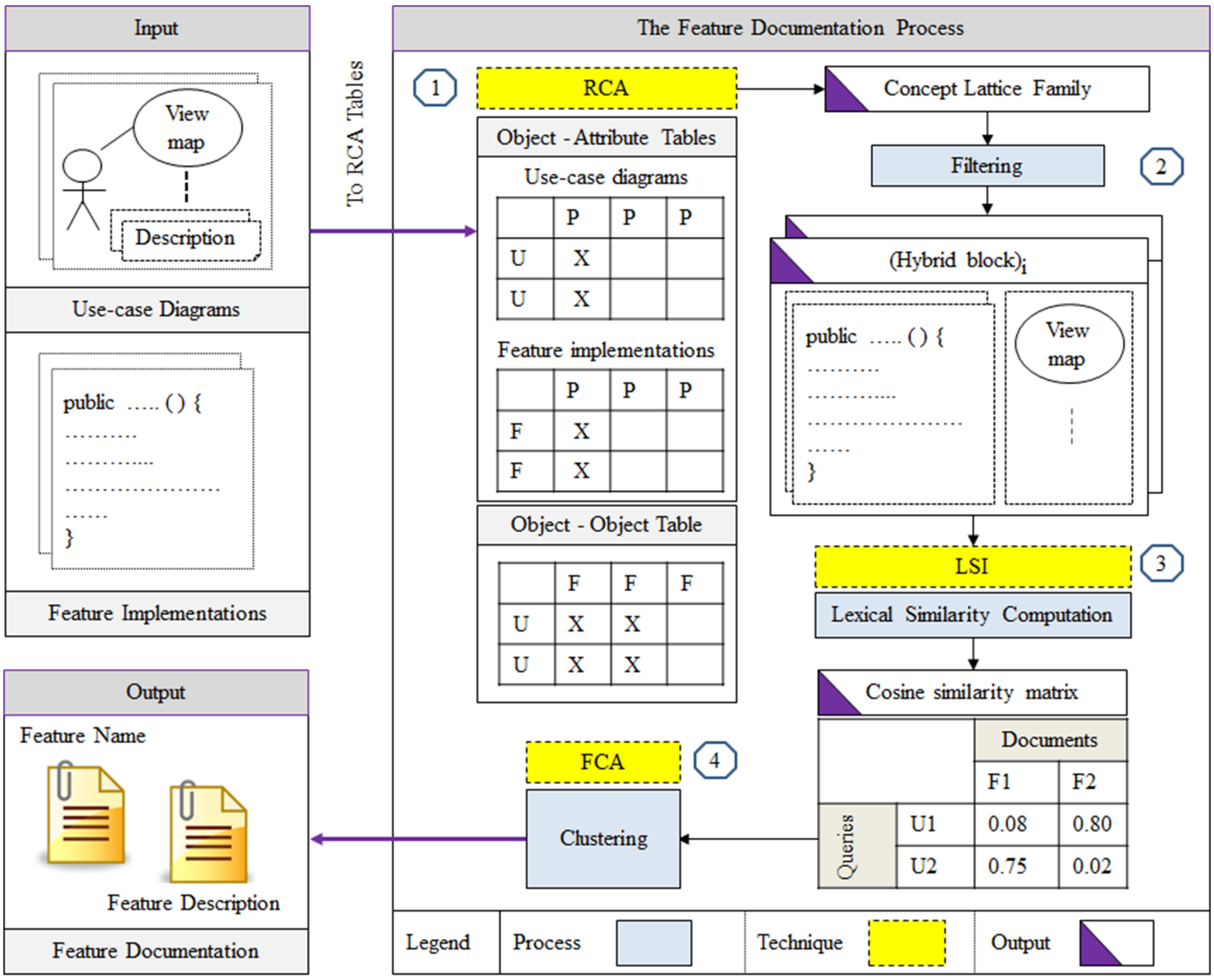
For a product variant, our approach takes as input a set of use-cases that the product variant supports and a set of mined feature implementations. Each use-case is identified by its *name* and *description.* A use-case description is given in a natural language. This information about the use-case represents a domain knowledge that is usually available from software variants documentation (*i.e.* requirement model). In our work, the use-case description consists of a short paragraph. For example, *retrieve data* use-case of Figure 5.3 is described in the following paragraph, "*the tourist can retrieve information and a small image of the attraction through his/her mobile phone. In addition, the tourist can store the current view of the map in the mobile phone*".

Our approach gives as output for each feature implementation, a name and description based on the use-case name and description. Each use-case is mapped into a functional feature based on our assumption. If there are two or more use-cases that have a relation with the single feature implementation, we consider all relevant use-cases as documentation for this feature implementation.

We take advantage of the *commonality* and *variability* across software variants for group feature implementations and the corresponding use-cases in the software family into disjoint, minimal clusters. As an example of the disjoint minimal cluster, the use-cases and feature implementations that are common to all software variants are grouped together as one cluster (*i.e.* minimal and disjoint search space). We called each disjoint minimal cluster a *hybrid block*. We exploit RCA technique to generate the reduced search spaces (*i.e.* hybrid blocks).

After reducing search space into a set of hybrid blocks, we rely on the textual similarity to identify, from each hybrid block, which use-cases depict the name and description of each feature implementation. We are exploiting LSI as lexical similarity method to link use-cases with the corresponding feature implementations based on the textual similarity. After identifying the textual similarity between use-cases and feature implementations, we group those elements as a set of clusters based on the textual similarity using FCA technique.

Figure 5.5 shows our feature documentation process. The feature documentation process takes the variants' use-cases and the mined feature implementations as its inputs. The first step (step ❶) of this process aims at identifying hybrid blocks based on RCA (*cf.* Section 5.5.1). The second step (step ❷) explores the concept lattice family to filter its hybrid blocks (*cf.* Section 5.5.2). In the third step (step ❸), we rely on LSI to determine the similarity between use-cases and feature implementations (*cf.* Section 5.5.3). This similarity measure is used to identify sets of clusters based on FCA (step ❹). Each cluster identifies the name and description for feature implementation (*cf.* Section 5.5.4).



	P	P	P
U	X		
U	X		

	P	P	P
F	X		
F	X		

	F	F	F
U	X	X	
U	X	X	

	F1	F2
U1	0.08	0.80
U2	0.75	0.02

Figure 5.5 : The feature documentation process.

5.5 Feature Documentation Step by Step

In this section, we describe the feature documentation process step by step. According to our approach, we identify the feature name and its description in three steps: ❶ identifying hybrid blocks of use-cases and feature implementations via RCA, ❷ measuring the lexical similarity between use-cases and feature implementations via LSI and ❸ identifying the feature name and its description via FCA as detailed in the following.

5.5.1 Reduce LSI Search Space: Identifying Hybrid Blocks Based on RCA

In this section we identify hybrid blocks of use-cases and feature implementations by using RCA technique. We use the existing use-case diagrams of software variants to document the mined feature implementations from these variants. In order to apply the LSI in an efficient way, we need to reduce the search space for use-cases and feature implementations. Starting from existing feature implementations and use-cases we cluster these elements into *disjoint minimal clusters* (*i.e.* hybrid blocks) to apply the LSI. Reducing the search space is based on the commonality and variability of software variants. To do so, we exploit commonality and variability across software variants at use-case and feature implementation levels.

RCA is used to cluster: the *common* use-cases and feature implementations among all software variants; the use-cases and feature implementations that are *shared* among a set of software variants, but not all variants; the use-cases and feature implementations that are *unique* for a single variant.

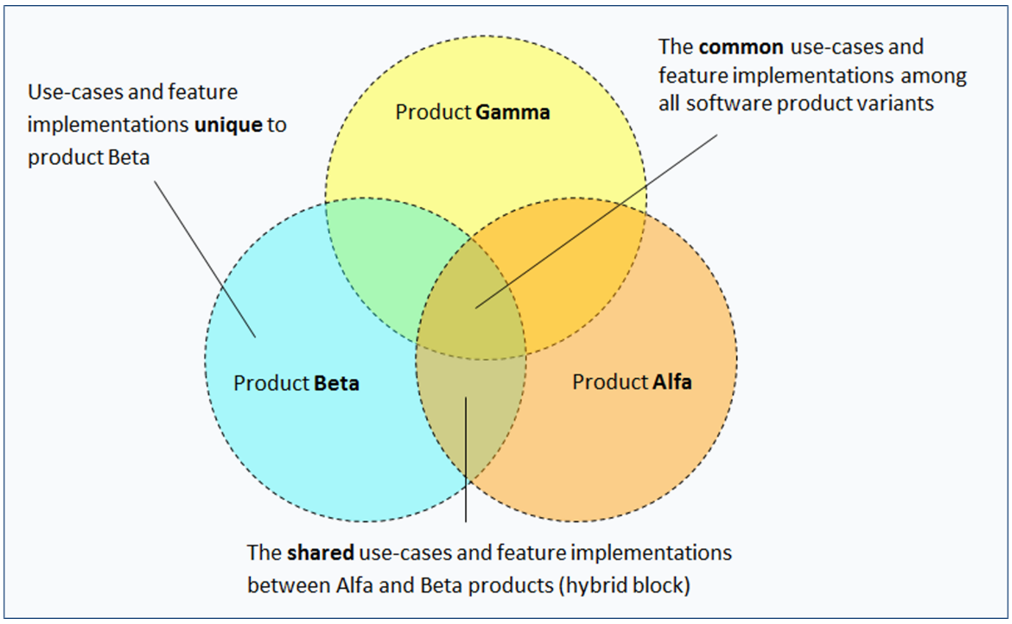

The **common** use-cases and feature implementations among all software product variants
Use-cases and feature implementations **unique** to product Beta
Product **Gamma**
Product **Beta**
Product **Alfa**
The **shared** use-cases and feature implementations between Alfa and Beta products (hybrid block)

Figure 5.6 : The common, shared and unique use-cases (*resp.* feature implementations) across software product variants.

The *Relational Context Family* (RCF) corresponding to our approach contains two formal contexts and one relational context, illustrated in Table 5.3. The first formal context represents the *use-case diagrams*. The second formal context represents *feature implementations*. In the formal context of *use-case diagrams*, the objects are use-cases and attributes are software variants. In the formal context of *feature implementations*, the objects are feature implementations and attributes are software variants. The relational context (*i.e. appear-with*) indicates which use-case appears with which feature implementation across software variants.

For RCF in Table 5.3, two lattices of the *Concept Lattice Family* (CLF) are represented in Figure 5.7. As an example of the hybrid block in Figure 5.7, we can see a set of use-cases (*cf.* $Concept\_1$ of the *Use_case_Diagrams* lattice) which always appears with a set of feature implementations (*cf.* $Concept\_6$ of the *Feature_Implementations* lattice) based on software configurations at use-case and feature implementation levels this is indicated by relational attribute *appears-with: Concept_6* in intent of *Concept_1*. As shown in Figure 5.7, RCA allows us to reduce the search space by exploiting commonality and variability across software variants.

RCF for feature documentation is generated automatically from use-case diagrams and the mined feature implementations of software variants[1]. In our work we consider RCA and FCA as clustering methods. RCA links each cluster of use-cases with the corresponding cluster of feature implementations by exploiting commonality and variability across software variants. The concept lattice family facilitates the comprehension of the clusters and their relations and provides a better visualization.

[1] `https://code.google.com/p/rcafca/`

Table 5.3 : The RCF for features documentation.

Use_case_Diagrams	MTG_1	MTG_2	MTG_3	MTG_4
View Map	✗	✗	✗	✗
Launch Google Map	✗	✗	✗	✗
View Direction	✗	✗	✗	✗
Show Street View	✗	✗	✗	✗
Place Marker on Map	✗	✗	✗	✗
Download Map		✗		
Show Satellite View			✗	
Show Next Attraction				✗
Search For nearest attraction				✗
Retrieve Data				✗

Feature_Implementations	MTG_1	MTG_2	MTG_3	MTG_4
Feature Implementation_1	✗	✗	✗	✗
Feature Implementation_2	✗	✗	✗	✗
Feature Implementation_3	✗	✗	✗	✗
Feature Implementation_4	✗	✗	✗	✗
Feature Implementation_5	✗	✗	✗	✗
Feature Implementation_6		✗		
Feature Implementation_7			✗	
Feature Implementation_8				✗
Feature Implementation_9				✗
Feature Implementation_10				✗

Relational context: appears-with	Feature Implementation_1	Feature Implementation_2	Feature Implementation_3	Feature Implementation_4	Feature Implementation_5	Feature Implementation_6	Feature Implementation_7	Feature Implementation_8	Feature Implementation_9	Feature Implementation_10
View Map	✗	✗	✗	✗	✗					
Launch Google Map	✗	✗	✗	✗	✗					
View Direction	✗	✗	✗	✗	✗					
Show Street View	✗	✗	✗	✗	✗					
Place Marker on Map	✗	✗	✗	✗	✗					
Download Map						✗				
Show Satellite View							✗			
Show Next Attraction								✗	✗	✗
Search For Nearest Attraction								✗	✗	✗
Retrieve Data								✗	✗	✗

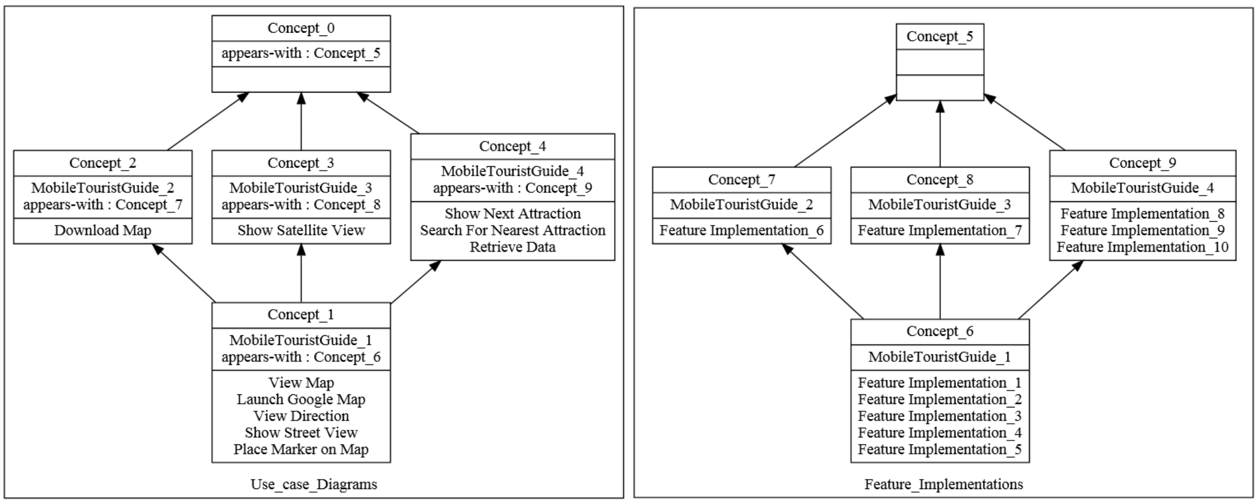


Figure 5.7 : The concept lattice family of relational context family in Table 5.3.

5.5.2 Exploring the Hybrid Blocks CLF to Identify Features Documentation

As concepts of the CLF are ordered, the search for hybrid blocks can be optimized if exploring the CLF from the bottom to the top block (from the more specific concept to the more general concept). We are exploring CLF and filtering them to get a set of hybrid blocks from bottom to top[2]. Figure 5.8 shows an example of the hybrid block resulting from filtering the CLF of Figure 5.7.

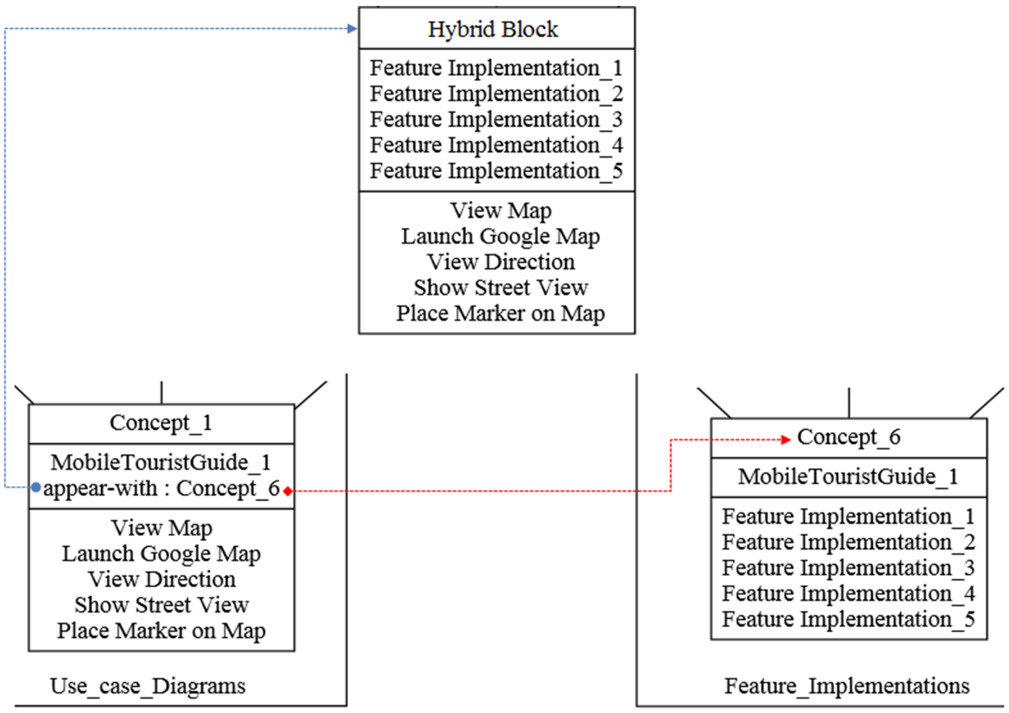


Figure 5.8 : Exploring and filtering the hybrid blocks CLF to identify features documentation.

As concepts of the concept lattice family are order, the search for hybrid block can be optimized if exploring the concept lattice family from bottom to top. For each concept in the Use_case_Diagrams lattice, we rely on the relational attribute appears-with: Concept_# (*e.g.* appears-with: Concept_9 in intent of concept_4 in Figure 5.7) to get the corresponding concept from the Feature_Implementations lattice. Then we construct the hybrid block based on the contents of both concepts (use-cases and feature implementations). For each recognized hybrid block we eliminate this concept from the filtering process. Then, we exploration of next (upper) concepts based on the relational attribute and so on for all concepts of the Use_case_Diagrams lattice. For instance, in the concept lattice family of MTG in Figure 5.7, the exploring and filtering process is performed as the follows: (Concept_1 – Concept_6); (Concept_2 – Concept_7); (Concept_3 – Concept_8); (Concept_4 – Concept_9); (Concept_0 – Concept_5).

5.5.3 Measuring Hybrid block contents' Similarity Based on LSI

In this section, we measure the lexical similarity between use-cases and feature implementations based on LSI. Based on the previous step, each hybrid block consists of a set of use-cases and a set of feature implementations. We need to identify from each hybrid block, which use-cases characterize the name and description for each feature implementation. To do so, we use the textual similarity between use-cases and feature implementations. This similarity measure is calculated using LSI. We rely on the fact that a use-case corresponding to the feature implementation is lexically closer to this feature implementation than to the rest of feature implementations.

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

To compute similarity between use-cases and feature implementation in the hybrid blocks, we proceed in three steps: ❶ building the LSI corpus, ❷ building the term-document matrix and the term-query matrix for each hybrid block and, lastly, ❸ building the cosine similarity matrix as detailed in the following.

5.5.3.1 Building the LSI Corpus

In order to apply LSI, we build a corpus that represents a collection of *documents* and *queries*. In our case, each use-case name and description in the hybrid block represents a *query* and each feature implementation represents a *document*. Our approach creates a query for each use-case. This query contains the use-case name and its description. Our approach also creates a document for each feature implementation. This document contains the OBE names (*cf.* Figure 5.9).

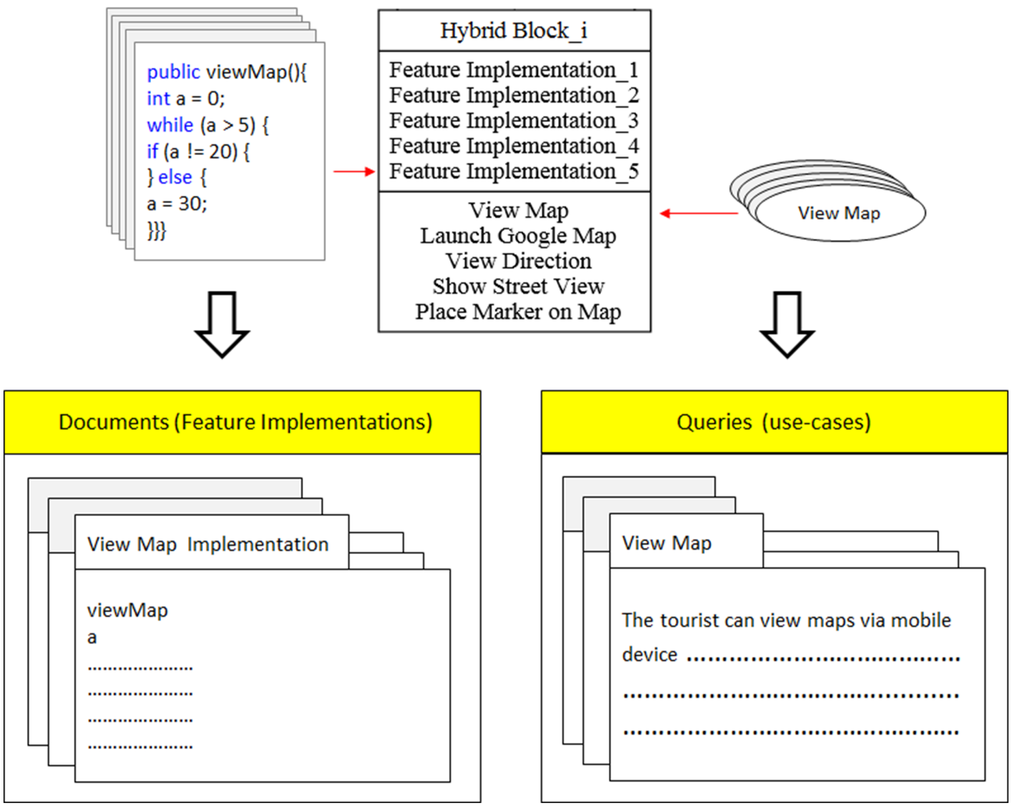


Figure 5.9 : Constructing a raw corpus from hybrid block.

Each feature implementation contains a set of OBEs such as packages, classes, attributes, methods or method body elements. Each feature implementation is represented by one document. Each document contains all OBE names that form this feature implementation. In our work, we store all the segments of OBE names, due to the importance of a complete OBE name. For example, for the OBE name *ManualTestWrapper*, all words are important {*manual, test and wrapper*}.

To be processed, the document and query must be normalized (*e.g.* all capitals turned into lower case letters, articles, punctuation marks or numbers removed). The normalized document is generated by analyzing the OBE names of feature implementation. All OBE names are split into terms and at last, word stemming is performed. The same procedure is followed to manipulate the use-case and its description to get the query document.

The most important parameter of LSI is the number of term-topics (*i.e.* k-Topics) chosen. A term-topic is a collection of terms that co-occur often in the documents of the corpus, for example {*user, account, password, authentication*}. Due to the nature of language use, the terms that form a topic are often semantically related. In our work, the number of *k-Topics* are equal to the number of *feature implementations* for each corpus.

5.5.3.2 Building the term-document and the term-query matrices for each hybrid block

All hybrid blocks are considered and the same processes applied to them. The *term-document matrix* is of size $m \times n$, where m is the number of terms extracted from feature implementations and n is the number of feature implementations (*i.e.* documents) in a corpus. The matrix values indicate the number of occurrences of a term in a document, according to a specific weighting scheme (*cf.* Table 5.4). The *term-query matrix* is of size $m \times n$, where m is the number of terms that are extracted from use-cases and n is the number of use-cases (*i.e.* queries) in a corpus. An entry of term-query matrix refers to the weight of i^{th} term in the j^{th} query. Table 5.4 shows the term-document and the term-query matrices of the hybrid block of *Concept_1* from Figure 5.7.

Table 5.4 : The term-document and the term-query matrices of *Concept_1* in Figure 5.7.

	Feature Implement._1	Feature Implement._2	Feature Implement._3	Feature Implement._4	Feature Implement._5
device	1	0	0	0	1
direction	0	0	0	6	0
google	1	0	0	0	0
launch	4	0	0	0	0
map	1	2	0	0	4
marker	0	6	0	0	0
mobile	1	0	0	0	1
place	0	3	0	0	0
show	0	0	2	0	0
street	0	0	5	0	0
tourist	1	1	1	1	1
view	0	0	1	2	5

➠ The term-document matrix

	Launch Google Map	Place Marker on Map	Show Street View	View Direction	View Map
device	1	0	0	0	1
direction	0	0	0	8	0
google	3	0	0	0	0
launch	3	0	0	0	0
map	2	2	1	1	5
marker	0	3	0	0	0
mobile	1	0	0	0	1
place	0	3	0	0	0
show	0	0	3	0	0
street	0	0	5	0	0
tourist	1	1	1	1	1
view	0	0	1	3	5

➠ The term-query matrix

In the term-document matrix, the *direction* term appears 6 times in the document *Feature Implementation_4*. In the term-query matrix, the *direction* term appears 8 times in the query *view direction*.

5.5.3.3 Building the cosine similarity matrix

Similarity between use-cases and feature implementations in each hybrid block is described by a cosine similarity matrix whose columns represent vectors of feature implementations and rows represent vectors of use-cases: documents as columns and queries as rows. The textual similarity between documents and queries is measured by the cosine of the angle between their corresponding vectors [Liu *et al.*, 2007].

The *k-Topics* for LSI in our approach are equal to the number of feature implementations in each hybrid block. In this example, K value is equal to 5. Table 5.5 shows the cosine similarity matrix of *Concept_1* (*i.e.* hybrid block) from Figure 5.7.

Table 5.5 : The cosine similarity matrix of *Concept_1* in Figure 5.7.

	Feature Implementation_1	Feature Implementation_2	Feature Implementation_3	Feature Implementation_4	Feature Implementation_5
Launch Google Map	0.861933577	0.0137010	0	0	0.152407
Place Marker on Map	0.01114798	0.9480070	0	0	0.085939
Show Street View	0.004088722	0.0051128	0.98581691	0.00571	0.070920
View Direction	0.00296571	0.0037085	0.0069484	0.999139665	0.108597
View Map	0.114676597	0.0627020	0.039159941	0.070025418	0.993111

In our work, we represent the similarity values between the use-cases and feature implementations. The results are represented as a directed graph. use-cases (*resp.* feature implementations) are represented as vertices and the similarity links as edges. The degree of similarity appears along the edges of the graph (*cf.* Figure 5.10). We should mention here that this graph just for visualization of the results.

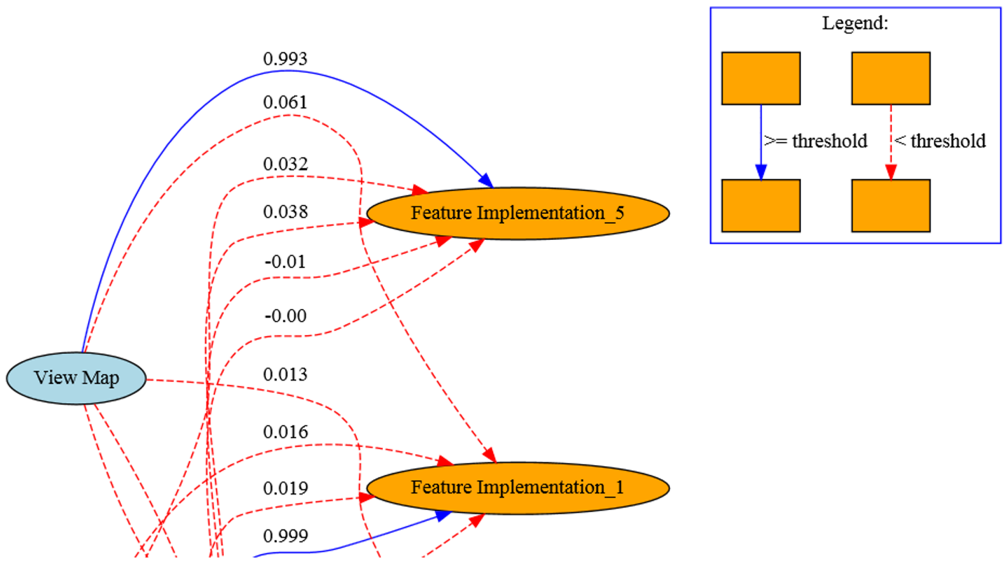


Figure 5.10 : The lexical similarity between use-cases and feature implementations as a directed graph.

5.5.4 Identifying Feature Name and Description Based on FCA

Based on the *cosine similarity matrix* we use FCA to identify, from each hybrid block of use-cases and feature implementations, which are similar elements. To transform the (numerical) cosine similarity matrices of the previous step into (binary) formal contexts, we use 0.70 as a threshold. 0.70 is currently used for cosine similarity (after testing many thresholds). This means that only pairs of use-cases and feature implementations having a calculated similarity greater than or equal to 0.70 are considered similar. Table 5.6 shows the formal context obtained by transforming the cosine similarity matrix corresponding to the hybrid block of *Concept_1* from Figure 5.7.

Table 5.6 : Formal context of *Concept_1* in Figure 5.7.

	Feature Implementation_1	Feature Implementation_2	Feature Implementation_3	Feature Implementation_4	Feature Implementation_5
Launch Google Map	✘				
Place Marker on Map		✘			
Show Street View			✘		
View Direction				✘	
View Map					✘

In this formal context, the use-case *"Launch Google Map"* is linked to the feature implementation *"Feature Implementation_1"* because their similarity equals 0.86, which is greater than the threshold. However, the use-case *"View Direction"* and the feature implementation *"Feature Implementation_5"* are not linked because their similarity equals 0.10, which is less than the threshold. The resulting AOC-poset (*cf.* Figure 5.11) is composed of concepts whose *extent* represents the use-case name and *intent* represents the feature implementation.

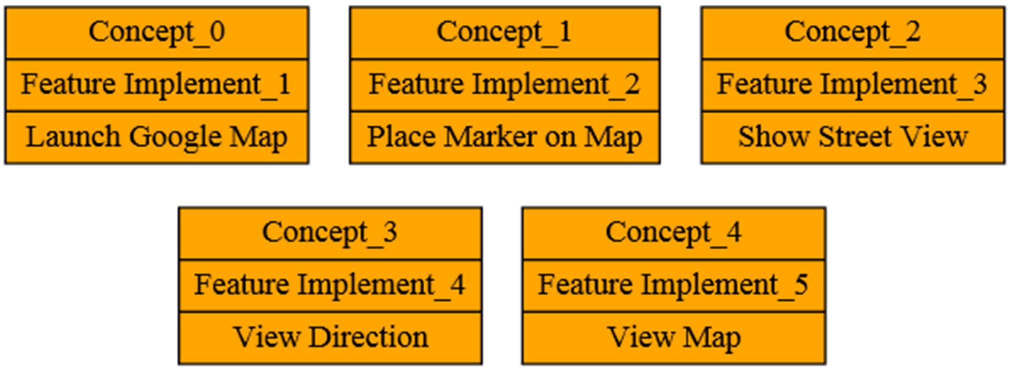


Figure 5.11 : The documented features from *Concept_1* in Figure 5.7.

For the MTG example, the AOC-poset of Figure 5.11 shows five non comparable concepts (that correspond to five distinct features) mined from a single hybrid block (*Concept_1* from Figure 5.7). The same feature documentation process is used for each hybrid block.

Table 5.7 shows the *product-by-feature matrix* that contains the mined and documented features from MTG software variants. This matrix is used as input to identify the FM in the next chapter.

Table 5.7 : The *product-by-feature matrix* for MTG software variants.

	View map	Place marker on a map	View direction	Launch Google map	Show street view	Download map from Google	Show satellite view	Search for nearest attraction	Show next attraction	Retrieve data
Mobile Tourist Guide 1	✗	✗	✗	✗	✗					
Mobile Tourist Guide 2	✗	✗	✗	✗	✗	✗				
Mobile Tourist Guide 3	✗	✗	✗	✗	✗		✗			
Mobile Tourist Guide 4	✗	✗	✗	✗	✗			✗	✗	✗

5.6 Naming Feature Implementation Based on OBE Names

In our approach we consider that a use-case diagram or other documentation (*i.e.* design documents) are not always available. In this case (*i.e.* when the design documents are missing) we rely only on the source code of software product variants to document the features. There are needs to extract feature name, especially from source code which is the most important source of information. In our work, we assume that the functional feature is implemented by a set of OBEs. Based on this fact, we use OBE names in each atomic block (*i.e.* feature implementations) to extract the feature name.

We rely on the same process proposed in [Kebir *et al.*, 2012]. Authors propose an approach to identify *components* from object-oriented source code of single software. Their work identifies component names based on *class names*. In our work, we assign *name* for each *feature implementation* based on its *OBE names*.

We identify the feature name in three steps: ① extracting and tokenizing OBE names of the identified feature implementation, ② weighting tokens and ③ constructing the feature name. We should mention that our approach can apply at any code granularity level (package, class, attribute, method, local variable, method invocation or attribute access) or on all levels.

❶ **Extracting and tokenizing OBE names from the identified feature implementation.** In this step, we rely on each feature implementation to extract the OBE names. Then, each OBE name is split into tokens according to the camel-case syntax. For example: *getMinimumSupport* is split into *get, Minimum* and *Support.* Camel-case technique is a simple and widely used method for identifier splitting algorithms [Dit *et al.*, 2011] and the rules of splitting are broadly based on Camel-case convention.

❷ **Weighting tokens.** In this step, a weight is assigned to each extracted token. A *large weight* (1.0) is given to tokens that are the first word of an OBE name. A *medium weight* (0.7) is given to tokens that are the second word of an OBE name. Finally a *small weight* (0.5) is given to the other tokens.

❸ **Constructing the feature name.** In this step, a feature name is constructed using the strongest weighted tokens. The strongest weighted token is the first word of the feature name; the second strongest weighted word is the second word of the feature name and so on.

The number of words used in the feature name is chosen by the *expert*. For example, the expert can select the top two words to construct the feature name. When many tokens have the same weight, all the possible combinations are presented to the expert and he can choose the appropriate one. Table 5.8 shows an example for constructing the feature name using *show street view* feature implementation of MTG software variants. In this example, the expert assigns a feature name based on the *top three tokens*. The proposed name for this feature implementation is *StreetShowView*.

Table 5.8 : OBE names, tokens, weight and strongest weighted tokens for *show street view* feature implementation.

	Token/Weight			
OBE Name	T1/ w=1.0	T2/ w=0.7	T3/ w=0.5	T4/ w=0.5
ShowStreetView	show	Street	View	
StreetPosition	Street	Position		
ChangeStreetSettings	Change	Street	Settings	
getStreetAddress	get	Street	Address	
setStreetAddress	set	Street	Address	
ShowNearestStreet	show	Nearest	Street	
ShowNextStreet	show	Next	Street	
retrieveStreetData	retrieve	Street	Data	
ShowStreet	show	Street		
updateStreetInfo	update	Street	Info	
ViewStreetMap	View	Street	Map	
ViewStreetPositionInfo	View	Street	Position	Info

Token	Total Weight	Top 3	Top 4
Show	4	✘	✗
Street	8	✘	✗
View	2.5	✘	✗
Position	1.2		✗
Change	1		
Settings	1		
get	1		
Address	1		
set	1		
Nearest	0.7		
Next	0.7		
retrieve	1		
Data	0.5		
update	1		
Info	1		
Map	0.5		

5.7 Threats to validity

This chapter presents the *second contribution* of REVPLINE approach: documenting the mined feature implementations. The documentation process is based on the use-case diagrams and feature implementations (*i.e.* OBEs). There are several threats to the validity of our approach.

❶ Developers might not use the same vocabularies to name OBEs and use-cases across software variants. This means that lexical similarity may be not reliable (or should be improved with other techniques) in all cases to identify the relationship between use-case and feature implementation.

❷ There is a limitation of using FCA as clustering technique. When we transform the (numerical) cosine similarity matrices into (binary) formal contexts, we use a threshold. So if the similarity value between query and document is greater than or equal the 0.70 the two documents are considered similar. By contrast, if the similarity value is less than the threshold (*i.e.* 0.69) the two documents are considered not similar. FCA deals with binary formal context (1, 0). This affects the quality of the result, since a similarity value 0.99 (*resp.* 0.69) is treated as a similarity value 0.70 (*resp.* 0).

❸ In our approach we consider that each use-case represents a functional feature. In some cases, two or more use-cases are grouped with single feature implementation; in this case we consider all relevant use-cases as documentation of this feature. This case should be improved with other techniques to extract unique name and description.

❹ Naming the feature based on the OBE names of feature implementation is not always reliable. In our approach, we rely on the top word frequencies (*i.e.* strongest weighted tokens) to propose the name for each implementation. The proposed name may be not relevant to the feature role. This means that OBE names are not reliable in all cases to identify feature name.

5.8 Conclusion

In this chapter, we proposed an approach for documenting the mined feature implementations of a set of software variants. We exploit commonality and variability across software variants at feature implementation and use-case levels to apply LSI method in an efficient way in order to document their features. Also we identify the feature name based on the OBE names. We consider the top OBE name frequencies as the feature name. Thus, we believe that REVPLINE offers a reasonable solution for naming/documenting the mined feature implementations from software variants. We recall the points that make our approach specific:

❶ REVPLINE accepts as inputs for the feature documenting process a set of feature implementations and use-case diagrams of software variants.

❷ REVPLINE approach offers two ways of naming the mined feature implementations: the first one is based on use-case diagrams of software variants and the second one relies on the OBE names.

❸ REVPLINE approach documenting the mined feature implementation and this process is very helpful. The feature documentation process aims at extracting name and/or description of features implementation. The purpose of this process is to understand, maintain and evolve the feature. In addition, for purposes of constructing an FM and reusing existing features in other software, each feature implementation that is presented to the human user must have a meaningful name.

Here, we compare the REVPLINE documentation approach to the related work. Table 5.9 shows an overview of the REVPLINE approach where we present the objectives, target systems, programmed method, type of code analysis, inputs, techniques, outputs and strategies.

Table 5.9 : Summary of REVPLINE: documenting the mined feature implementation.

Objectives				Software		Programmed Method		Code Analysis		Input		Strategies	Techniques				Output		Language	Case study			Tool support
Feature documentation	Software maintenance	re-engineering	Code-to-document traceability link	Single software	Software family	Automatic	Semi-automatic	Static	Dynamic	Feature implementations (*i.e.* OBEs)	Use-case diagram	Reduce search space	FCA	RCA	LSI	Automatic heuristic	Feature name	Feature description	Java	Small	Medium	Large	
✘	✘	✘	✘		✘		✘	✘		✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘

CHAPTER 6

Reverse Engineering Feature Models from Software Configurations

Fundamental progress has to do with the reinterpretation of basic ideas.
Alfred North WHITEHEAD

Preamble

In this chapter, we present the third contribution of the REVPLINE; which is reverse engineering feature models from software configurations. Section 6.1 gives an introduction of this chapter. It also presents our goal and motivation. In addition this section gives an example of FM and its software configurations. Then, Section 6.2 gives an overview of the reverse engineering FM process. Next, Section 6.3 presents the reverse engineering FM process step by step. Section 6.4 presents the way that we use to evaluate the obtained FMs by our approach. It also discusses threats to the validity of the reverse engineering FMs process. Finally in Section 6.5, we conclude this chapter.

6.1 Introduction

As stated previously, the main goal of this dissertation is to re-engineering software product variants into SPL. In fact, we contribute in some important phases of this process such as feature location, feature documentation and identify the dependencies between the mined and documented features. In the previous two chapters of this dissertation, we have presented an approach to identify features from object-oriented source code of a collection of software product variants and document them. Dependencies between features need to be expressed via FM. Based on product-by-feature matrix, which contains the mined and documented features from software variants, we propose an approach to reverse engineering FMs from software configurations by using FCA. This chapter focuses on this goal.

Even for a small set of configurations, manual construction of a FM is time-consuming and error-prone [Davril *et al.*, 2013] [Acher *et al.*, 2013a]. Thus, in this chapter, we propose an approach to reverse engineering FMs from the mined and documented features of software variants. We rely on FCA and software configurations to identify FMs. The mined FM defines all the valid feature configurations.

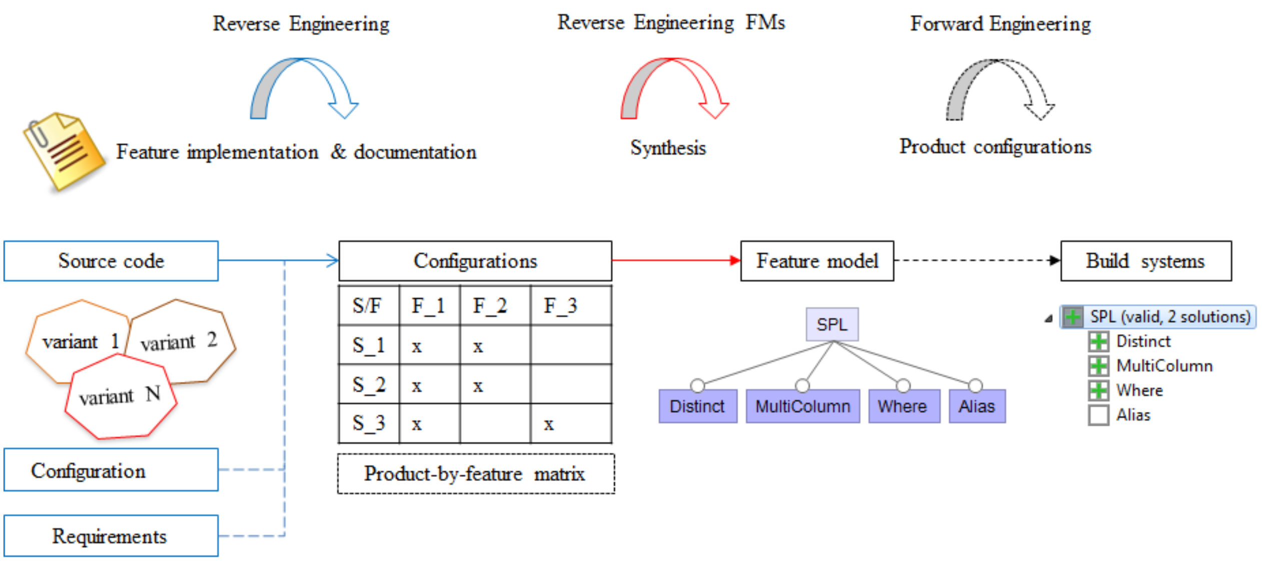


Figure 6.1 : From configurations to a feature model.

For obtaining such a FM, mandatory and optional features for software product variants have to be identified and documented. The mined and documented features are sorted as product-by-feature matrix. This matrix represents the software configurations (*cf.* Figure 6.1). Then, we rely on FCA to mine a unique and consistent feature model.

Figure 6.2 shows the feature model of the cell phone SPL. We rely on these product configurations and on the existing FM to illustrate the remaining of this chapter.

We consider the 16 valid product configurations (*cf.* Table 6.1) defined by the FM in Figure 6.2. Using the FAMA tool suite[1] [Benavides *et al.*, 2010], we computed all these valid product configurations for cell phone FM and use them as an input to our approach (*cf.* Section 6.4). We rely on these configurations and on the existing FM to illustrate our approach.

[1]FAMA Tool Suite : http://www.isa.us.es/fama/

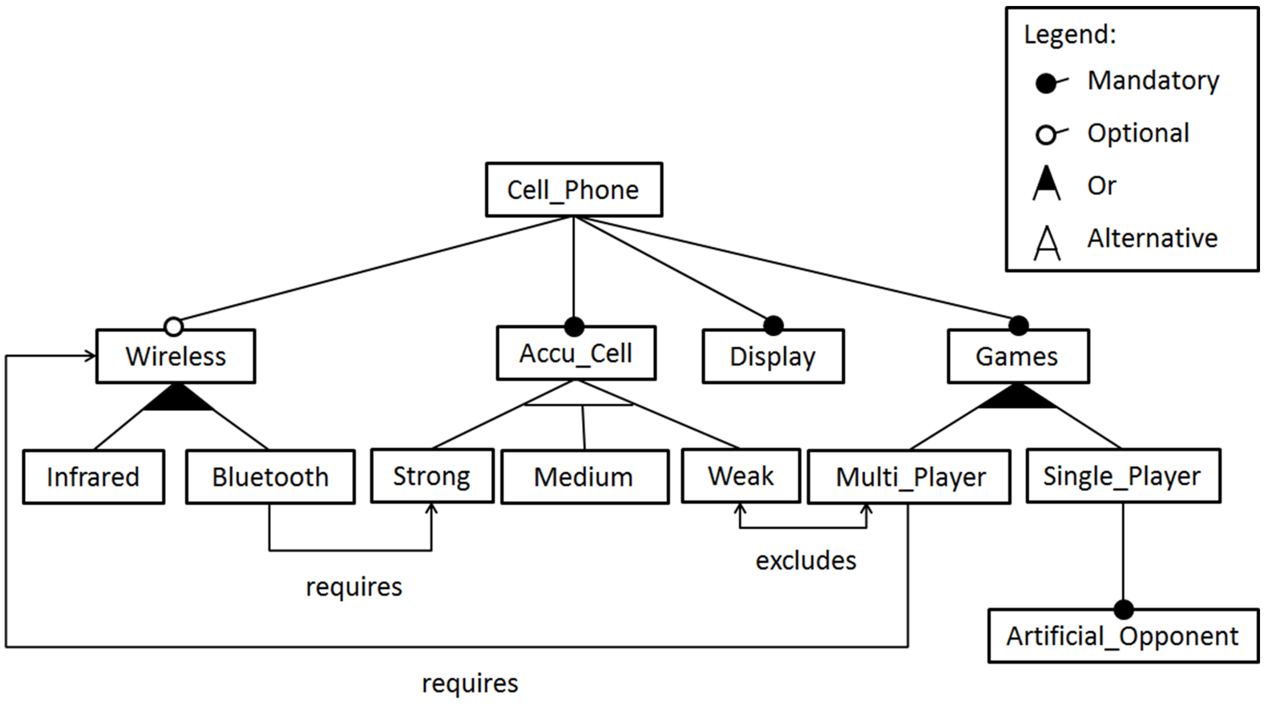


Figure 6.2 : Cell phone SPL feature model [Haslinger, 2012].

Table 6.1 : Valid product configurations of cell phone SPL [Haslinger, 2012].

Product configurations	Cell_Phone	Wireless	Infrared	Bluetooth	Accu_Cell	Strong	Medium	Weak	Display	Games	Multi_Player	Single_Player	Artificial_Opponent
Product-1	✗	✗	✗		✗	✗			✗	✗	✗		
Product-2	✗	✗		✗	✗	✗			✗	✗	✗		
Product-3	✗	✗	✗	✗	✗	✗			✗	✗	✗		
Product-4	✗	✗	✗		✗		✗		✗	✗	✗		
Product-5	✗				✗			✗	✗	✗		✗	✗
Product-6	✗				✗	✗			✗	✗		✗	✗
Product-7	✗	✗	✗		✗	✗			✗	✗		✗	✗
Product-8	✗	✗		✗	✗	✗			✗	✗		✗	✗
Product-9	✗	✗	✗	✗	✗	✗			✗	✗		✗	✗
Product-10	✗				✗		✗		✗	✗		✗	✗
Product-11	✗	✗	✗		✗		✗		✗	✗		✗	✗
Product-12	✗	✗	✗		✗			✗	✗	✗		✗	✗
Product-13	✗	✗	✗		✗		✗		✗	✗	✗	✗	✗
Product-14	✗	✗	✗		✗	✗			✗	✗	✗	✗	✗
Product-15	✗	✗		✗	✗	✗			✗	✗	✗	✗	✗
Product-16	✗	✗	✗	✗	✗	✗			✗	✗	✗	✗	✗

Product-by-feature matrix
(✗ feature is in the product; otherwise feature is not in the product)

6.2 FM Reverse Engineering Process

FMs are one of the most popular formalisms for modeling and reasoning about commonality and variability of an SPL. FMs are used extensively in SPL to help generate and validate individual product configurations and to provide support for domain analysis [Davril *et al.*, 2013]. In our work, we mine feature models from the concrete features that are mined and documented by our previous work. The aim of the mined FM is to represent all dependencies between the concrete features.

In the first step of our approach, software variant features are mined from software variants source code. In the second step, these software features are documented in order to generate a product-by-feature matrix $P \times F$ in which the rows of the matrix correspond to products and the columns correspond to features. The $(i, j)^{th}$ entry of this matrix can take a value of 0 or 1, to represent whether the i^{th} product is known to include the j^{th} feature or not.

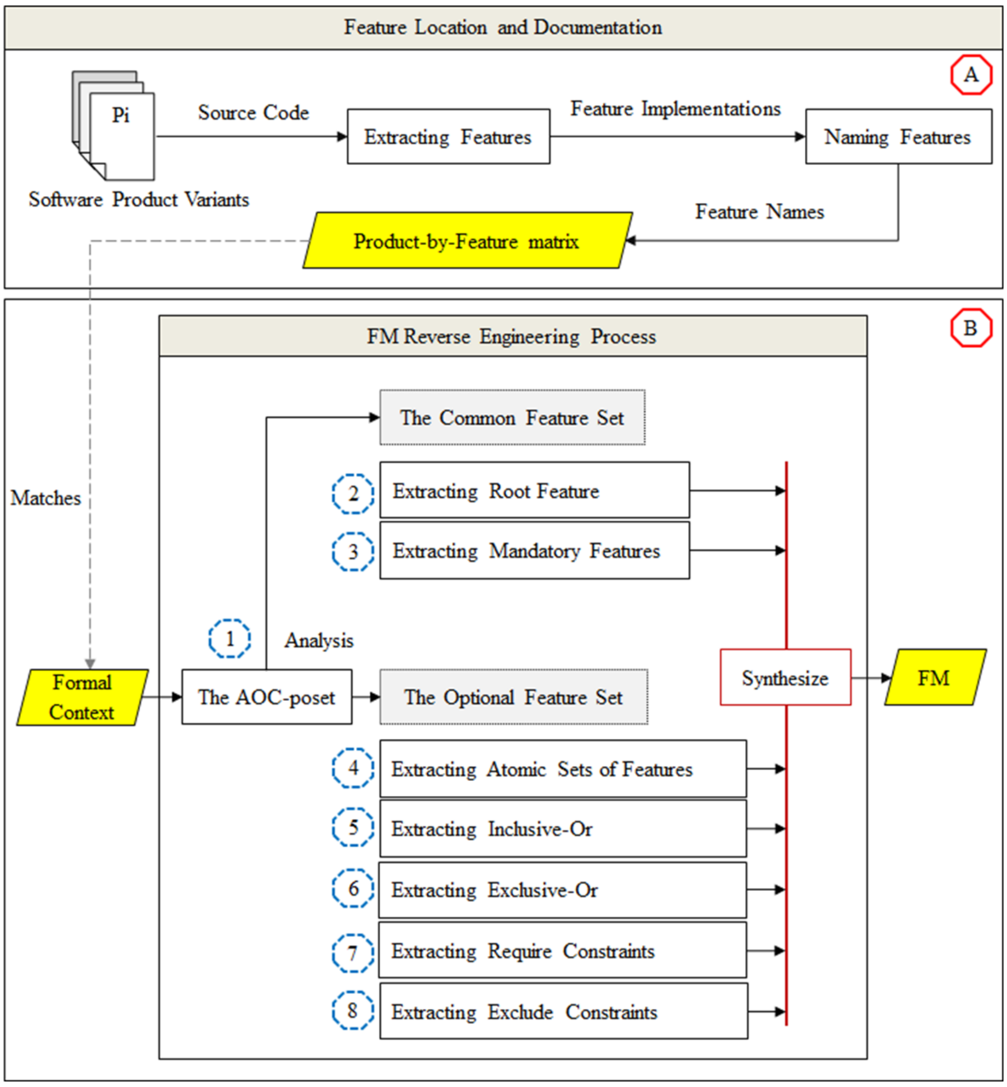


Figure 6.3 : FM reverse engineering process.

Software product variants consist of 2 or more software. These variants consist of N features. As the number of features is high, reverse engineering FMs entails to reduce this search space. Several strategies can be combined to do so: Separate the features set into two subsets, the common feature set (*i.e.* mandatory features) and the optional feature set (*i.e.* optional features), on which the same search process will have to be performed. The optional (*resp.* common) features appear in some but not all (*resp.* all) variants. Next, separate the optional feature set into small subsets that each contains features shared by groups of two or more variants or features that are hold uniquely by a given variant. Then, identify all kinds of relations between the optional features based on AOC-poset structure. Then, identify FM as a tree-like hierarchy of features and constraints between them.

Figure 6.3 shows our FM reverse engineering process. In this figure, our goal is represented in the bottom part (*i.e.* part B). The top part (*i.e.* part A) of this figure has been processed already in our previous chapters. Part B of this figure shows the steps of the FM reverse engineering process. Based on the *product-by-feature* matrix, we reverse engineer a FM by using FCA, as follows: ➊ extracting the AOC-poset from formal context (product-by-feature matrix), ➋ extracting the root feature of the FM, ➌ extracting mandatory features, ➍ extracting atomic sets of features (optional AND-groups of features), ➎ extracting Inclusive-or relations, ➏ extracting Exclusive-or relations, ➐ extracting Require constraints, and ➑ extracting Exclude constraints.

6.3 Step-by-Step FM Reverse Engineering

In our work, the root feature appears in all software variants. The mandatory features appear in all configurations. From this, we can notice that all mandatory features and the root of the feature model are appearing in all valid product configurations that are defined by the FM. The remaining features represent the optional features; these features do not appear in all valid product configurations. On the other hand, the optional feature may be required by another feature or excludes another one. The constraints between these optional features are very important for the mined feature model. This section presents step-by-step the FM reverse engineering process. According to our approach, we identify FM in eight steps as detailed in the following.

6.3.1 Extracting the AOC-poset

The first step of our FM extraction process is the identification of the AOC-poset. First, a *formal context*, where objects are software product variants and attributes are features (*cf.* Table 6.1), is defined. The corresponding *AOC-poset* is then calculated. The intent of each concept represents features common to two or more products or unique for one product. As concepts of AOC-posets are ordered, the intent of the most general (*i.e.* top) concept gathers mandatory features that are common to all products. The intents of all remaining concepts represents the optional features. The extent of each of these concepts is the set of products having these features in common (*cf.* Figure 6.4).

Algorithm 1 is a simple algorithm for building the Hasse diagram of the AOC-poset. In this algorithm, we use complementary classical FCA notations: for any object set $S_o \subseteq O$, the set of shared attributes is $S'_o = \{a \in A | \forall o \in S_o, (o,a) \in R\}$, and for any attribute set $S_a \subseteq A$, the set of owners is $S'_a = \{o \in O | \forall a \in S_a, (o,a) \in R\}$. In the following algorithms, for a Concept C, we will denote by intent (C), extent (C), simplified intent (C), and simplified extent (C) its associated sets.

The AOC-poset is a structured representation of the products that contains many information about the relationships between features, between products and between features and products. Nevertheless, extracting a FM from this structure is not direct and has many acceptable solutions. To give a simple example, from the AOC-poset, when a feature f1 is introduced in a sub-concept of the concept which introduces a feature f2, we can infer either that f1 is a sub-feature of f2, or that f1 requires f2, or that this is . Besides, as the initial set of product configuration may not be exhaustive, we mainly look for a FM which recognizes all initial configurations and serves as a basis for an expert engineer to design a final FM.

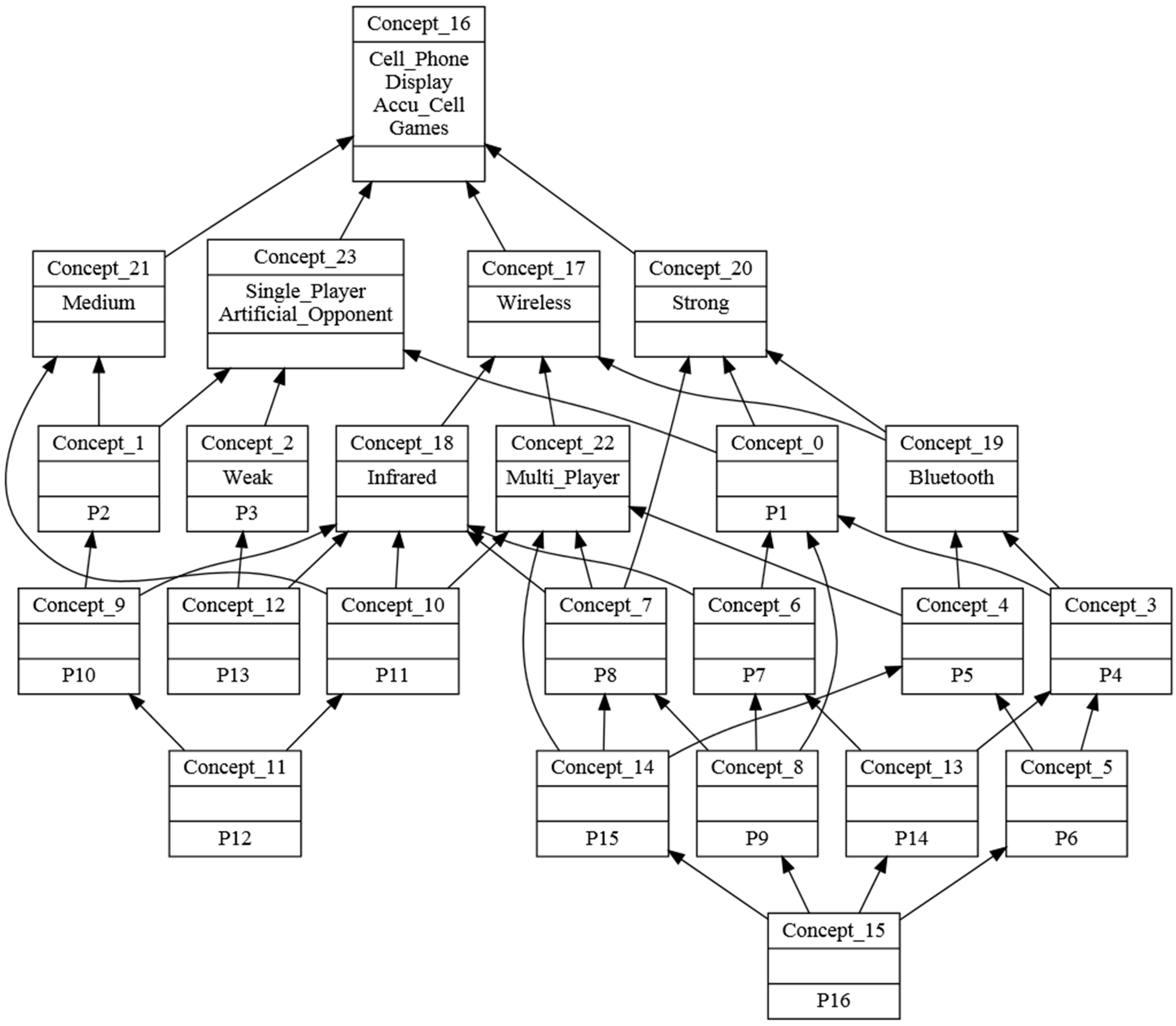


Figure 6.4 : The AOC-poset for the formal context of Table 6.1.

Algorithm 1: ComputeAOCposet (K)

```
Data: K: a formal context; K = (O, A, R) where O and A are sets (product and feature respectively)
      and R is a binary relation.
Result: $(AOC_K, \leq_s)$: the AOC-poset associated with K
// compute the object concepts and the attribute concepts
$AOC_K \leftarrow \emptyset$
foreach $o \in O$ do
    $AOC_K \leftarrow AOC_K \cup (\{o\}'', \{o\}')$ // that is, objects that share the same attributes as o, with the
    attributes of o
foreach $a \in A$ do
    $AOC_K \leftarrow AOC_K \cup (\{a\}', \{a\}'')$ // that is, objects that share the attribute a, with the attributes
    they share
// establish the specialization order
Compute the transitive reduction of $\leq_s$ by comparing the concept extents in $AOC_K$ with inclusion
```

6.3.2 Extracting root feature

The root feature of FM in SPL usually refers to the software family name. Feature *Cell_Phone* is the root feature of cell phone FM; hence it is selected in every program configuration. Features appearing at the top concept in the AOC-poset are used in every product configuration (*cf.* Figure 6.5). In our work, we select the feature in the top concept which represents the product family (*i.e. Cell_Phone*) to be the root of the FM.

6.3.3 Extracting mandatory features

Mandatory features appearing at the top concept in the AOC-poset are used in every product configuration. These features are likely to be mandatory features, *i.e.* they need to be included in every product. In our work, we select all the features in the top concept (*i.e.* Accu_Cell, Display, Games) to be the mandatory features of the FM except the root feature (*i.e. Cell_Phone*). Figure 6.5 shows the identify mandatory features. Algorithm 2 is a simple algorithm for building Base feature.

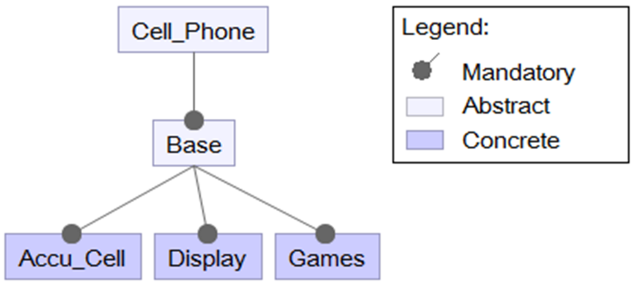


Figure 6.5 : The mandatory features (*i.e.* base group of features) identified from the cell phone product configurations.

Algorithm 2: ComputeRootAndMandatoryFeature

```
// Top concept ⊤
∃ F ∈ A, which represents the name of the software family with F in feature set of ⊤
Data: $AOC_K$, $\leq_s$: the AOC-poset associated with $K$
Result: part of the FM containing root and mandatory features
// Compute the root Feature
CFS ← intent (⊤)
Create node root, label (root) ← F, type (root) ← abstract
CFS' ← CFS \ {F}
if CFS' ≠ ∅ then
    Create node base with label (base) ← "Base"
    type (base) ← abstract
    Create edge e = (root, base)
    type (e) ← mandatory
    for each $F_e$ in CFS' do
        Create node feature, with label (feature) ← $F_e$
        type (feature) ← concrete
        create edge e = (base, feature)
        type (e) ← mandatory
```

6.3.4 Extracting atomic set of features (AND-group)

An atomic set of features groups features that always appear together in product configurations. When two optional features appear in the same simplified intent, this means that these features are always used together in all product configurations of the concept extent. These are likely to be features that cannot be used separately. Such a concept corresponds to an AND-group of features. If two features F_1 and F_2 are introduced in the same concept, this means that they co-occur and always appear together which gives rise to the equivalence $F_1 \leftrightarrow F_2$. For our illustrative example, the AOC-poset of Figure 6.4 shows a simplified intent with two features *Single_Player* and *Artificial_Opponent*. These features are likely to form an atomic set of features (*cf. Concept_23* from Figure 6.4). In Figure 6.6, the two forms are equivalent; the two features in all cases appear together. Algorithm 3 is a simple algorithm for building AND-group of features.

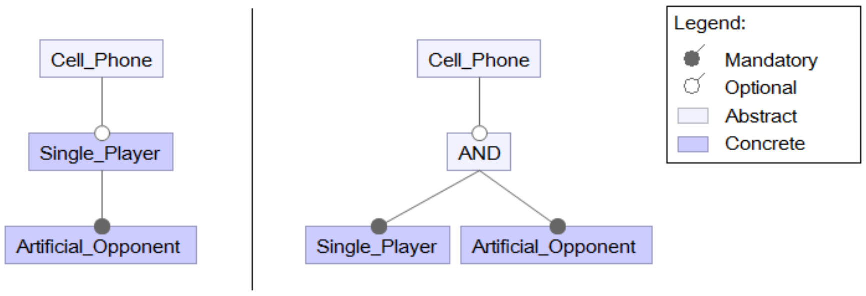


Figure 6.6 : An atomic group of features (AND-group) identified from the cell phone product configurations.

Algorithm 3: Compute*AtomicSetOfFeatures* (and groups)

```
Data: $AOC_K$, $\leq_s$: the AOC-poset associated with $K$
Result: part of the FM with and groups of features
// Compute atomic set of features
// Feature List (FL) is the list of all features (FL = A in K=(O, A, R)).
FL' ← FL \ CFS // FL \ intent (⊤)
AsF ← ∅
int count ← 1
for each concept $C \neq \top$ such that | simplified intent ($\mathscr{C}$) | ≥ 2 do
    AsF ← AsF ∪ simplified intent ($\mathscr{C}$)
    Create node and with label (and) ← "AND"+ count
    type (and) ← abstract
    create edge $e$ = (root, and)
    type ($e$) ← optional
    for each F in simplified intent ($\mathscr{C}$) do
        create node feature, with label (feature) ← $F$
        type (feature) ← concrete
        create edge $e$ =(and, feature)
        type ($e$) ← mandatory
```

6.3.5 Extracting exclusive-or relation

Features that form exclusive-or relation can be recognized in the concept lattice using the meet (denoted by ⊓) lattice operation [Loesch and Ploedereder, 2007], or computing the greatest lower bounds in the AOC-poset. If a feature A is introduced in concept C_1, a feature B is introduced in concept C_2 and $C_1 \sqcap C_2 = \bot$ (and extent($\bot$) = $\emptyset$), that is, if the bottom of the lattice is the greatest lower bound of C_1 and C_2, the two features never occur together in a product. These features are likely to be alternative features, *i.e.* they always have to be used mutually exclusively. In our example, in the AOC-poset of Figure 6.4, features *Strong*, *Medium*, and *Weak* form an exclusive-or relation (*cf.* Figure 6.7). The corresponding concepts (*i.e.* Concept_21, Concept_20, and Concept_2) don't have a common lower bound in the AOC-poset. Algorithm 4 is a simple algorithm for building an Xor group of features. The principle is to compare the super-concepts sets of all the minimum elements of the AOC-poset and to keep concepts that are not super-concepts of two (or more) minimum concepts.

Algorithm 4: Compute*Exclusive-or*Relation (Xor)

```
Data: $AOC_K$, $\leq_s$: the AOC-poset associated with $K$
Result: part of the FM with XOR group of features
// Compute exclusive-or relation
FL'' ← FL' \ AsFs
$\mathscr{C}xor \leftarrow \emptyset$
SSCS ← ∅ // set of super-concept sets
Minimum-set ← ∅
for each minimum of $AOC_K$ denoted by m do
    Let SSC the set of super-concepts of m (except ⊤)
    SSCS ← SSCS ∪ {SSC}
    Minimum-set ← Minimum-set ∪ {m}
    $\mathscr{C}xor \leftarrow \mathscr{C}xor \cup$ SSC
while SSCS ≠ ∅ do
    SSC-1 ← any element in (SSCS)
    SSCS ← SSCS \ SSC-1
    for each SSC-2 in SSCS do
        $\mathscr{C}xor \leftarrow \mathscr{C}xor \setminus$ (SSC-1 ∩ SSC-2)
XFS ← ∅
if $|\mathscr{C}xor| > 1$ then
    Create node xor with label (xor) ← "XOR"
    type (xor) ← abstract
    create edge e = (root, xor)
    // if all products are covered by $\mathscr{C}xor$
    if $\cup_{C \in \mathscr{C}xor} extent(C) = O$ then
        type (e) ← mandatory
    else
        type (e) ← optional
    for each concept $C \in \mathscr{C}xor$ do
        for each F in simplified intent ($\mathscr{C}$) do
            create node feature, with label (feature) ← F
            type (feature) ← concrete
            create edge e = (xor, feature)
            type (e) ← alternative
            XFS ← XFS ∪ F
```

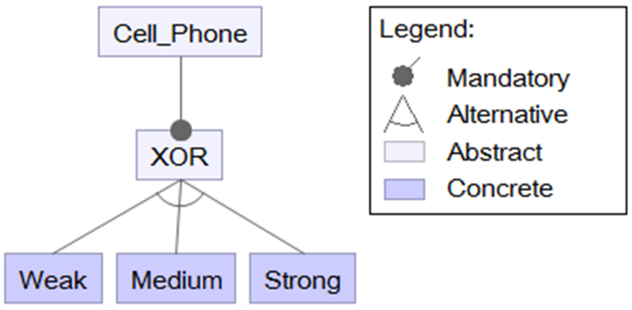


Figure 6.7 : Xor group of features identified from the cell phone product configurations.

6.3.6 Extracting inclusive-or relation

Optional features are used in some product configuration, *i.e.* they don't need to be included in every product. If we prune the AOC-poset by removing the top concept, concepts that represent an atomic set of features, and concepts that represent features that form exclusive-or relation, the remaining concepts represent features that forms inclusive-or relation. In the AOC-poset of Figure 6.4, features *Wireless*, *Infrared*, *Bluetooth*, and *Multi_Player* form an inclusive-or relation (*cf.* Figure 6.8). Algorithm 5 is a simple algorithm for building the *Or* group of features.

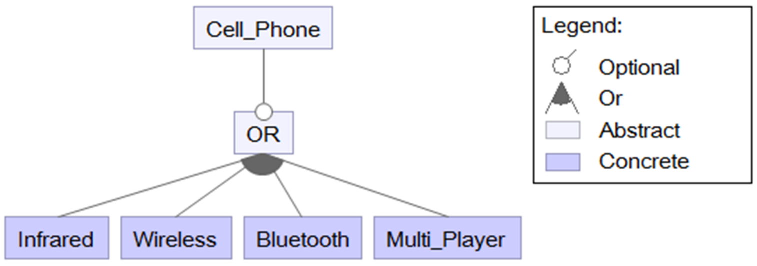


Figure 6.8 : Or group of features identified from the cell phone product configurations.

Algorithm 5: ComputeInclusive-orRelation (Or)

```
Data: $AOC_K, \leq_s$: the AOC-poset associated with $K$
Result: part of the FM with OR group of features
// Compute inclusive-or relation
FL''' ← FL'' \ XFS
if $FL''' \neq \emptyset$ then
    Create node or with label (or) ← "OR"
    type (or) ← abstract
    create edge e = (root, or)
    type (e) ← optional
    for each F in FL''' do
        create node feature, with label (feature) ← F
        type (feature) ← concrete
        create edge e = (or, feature)
        type (e) ← Or
```

6.3.7 Extracting requires constraint

Require constraint, *e.g.* saying "variable feature A always requires variable feature B", can be extracted from the lattice via forward implications (upward paths in the lattice). We say that A implies B (written $A \rightarrow B$). The required relations can be identified in the AOC-poset via implication rules: when a feature F_1 is introduced in a subconcept of the concept that introduces another feature F_2, there is an implication: $F_1 \rightarrow F_2$. In the AOC-poset of Figure 6.4, there are six Require constraints: Infrared → Wireless; Bluetooth → Wireless; Bluetooth → Strong; Multi_Player → Wireless; Weak → Artificial_Opponent; Weak → Single_Player. Remark that implications ending to mandatory features are useless because they are represented in the FM by the Base node. Algorithm 6 is a simple algorithm for identifying Require constraints. It works on the transitive reduction of the AC-poset, which is the AOC-poset restricted to Attribute concepts (with non empty simplified intents). Some of these requires constraints, when there are among the children of the OR node, like Infrared → Wireless; Bluetooth → Wireless, could be inserted as sub-features (Infrared and Bluetooth being sub-features of Wireless) as an improvement of this approach.

Algorithm 6: Compute*Require*Constraint ($Requires$)

```
Data: $AC_K$, $\leq_s$: the AC-poset associated with $K$
Result: Require - the set of require constraints
Require ← ∅
for each edge (C1, C2) = e in transitive reduction of AC-poset do
    for all f1, f2 with f1 ∈ simplified intent (C1) and f2 ∈ simplified intent (C2) do
        Require ← Require ∪ {f1 ⟶ f2}
```

6.3.8 Extracting Exclude constraint

To mine excludes constraints from AOC-poset, we use the meet of the introducers of the two involved features. For example, the meet of Concept_2 which introduces *Weak* and Concept_22 which introduces Multi_Player is the bottom (in the whole lattice). In the AOC-poset they don't have common lower bound. We can thus deduce that $\neg(Weak \wedge Multi\_Player)$. In the AOC-poset of Figure 6.4, there are three excludes constraints: ¬ (Multi_Player ∧ Weak); ¬ (Bluetooth ∧ Medium); ¬ (Bluetooth ∧ Weak). Algorithm 7 is a simple algorithm for identifying Exclude constraints. For extracting them, we compare features that are below the OR group with each set of features in the intent of a minimum, in order to determine which are incompatible: this is the case for a pair (f1, f2) where f1 is in the OR group and not in the minimum intent, and f2 is in the minimum intent but not in the OR group.

Algorithm 7: Compute*Exclude*Constraint ($Excludes$)

```
Data: $AOC_K$, $\leq_s$: the AOC-poset associated with $K$
Result: Exclude - the set of exclude constraints.
// Minimum-set from Algorithm 4
// FL''' from Algorithm 5
Exclude ← ∅
for each $P \in$ Minimum-set do
    $Pintent \leftarrow intent(P) \setminus intent(\top)$
    Opt-feat-set ← $FL''' \setminus (FL''' \cap Pintent)$
    Super-feat-set ← $Pintent \setminus (FL''' \cap Pintent)$
    if Opt-feat-set ≠ ∅ and Super-feat-set ≠ ∅ then
        for each f1 ∈ Opt-feat-set, f2 ∈ Super-feat-set do
            Exclude ← Exclude ∪ {¬(f1 ∧ f2)}
```

6.3.9 The Resulting Feature Model

Figure 6.9 shows the resulting FM based on the product configurations of Table 6.1. This FM consists of a root feature, base group of features (mandatory features), atomic sets of features (and-groups; in the general case we can have more than one), xor and or group of features, and require and exclude constraints. The resulting FM describes all of the product configurations that are defined by the initial FM (*cf.* Section 6.4). REVPLINE generates thus a unique, consistent, maintainable and meaningful FM.

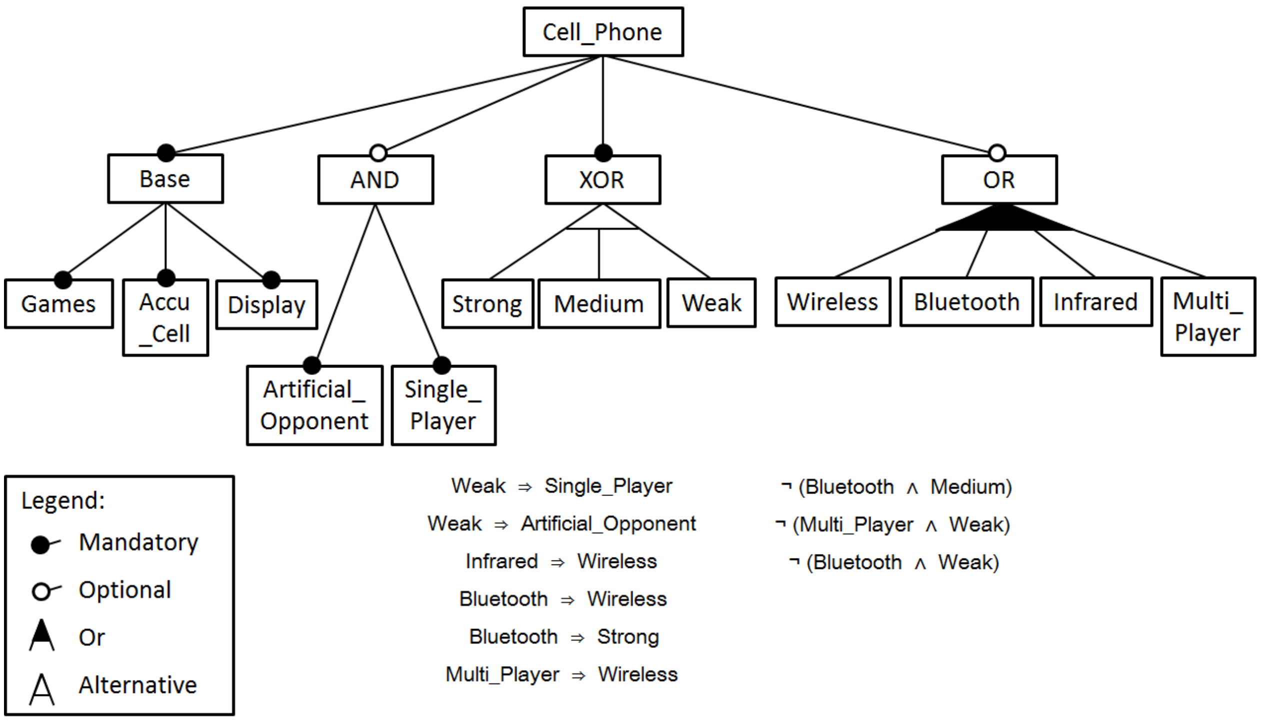


Figure 6.9 : The resulting FM based on the product configurations of Table 6.1.

6.4 FM Evaluation

We performed an evaluation of our approach using the *cell phone SPL FM*. The reasons behind this choice are: the selected FM is correct (valid product configurations) and contains all basic elements of the FM in addition to the cross-tree constraints (CTCs). The cell phone SPL FM has been used and evaluated in [Haslinger, 2012]. The number of used product configurations in this example is equal to 16 and the number of all features is equal to 13.

As semantics of an FM is given by its configuration set, we compare the sets of configurations defined by the two FMs (*i.e.* the initial FM/mined FM) and we discuss the *recall*, *precision* and *F-Measure* metrics for the mined product configurations (considering that the initial FM, manually designed FM, is correct). This section explains our approach to evaluate the obtained FMs by using our approach.

In order to evaluate the mined FM we rely on the SPLOT homepage and the FAMA Tool Suite. The goal of using the FM editor of the SPLOT homepage and produce the FM in SXFM format is to compare the two FMs (the initial FM/mined FM). SXFM[2] is for simple XML FM format used in SPLOT homepage. The SXFM format can be used online through SPLOT's feature model editor. To get all information from SXFM file, Java parser is available for reading the SXFM file at SPLOT homepage. The SXFM format for the mined cell phone FM (in addition to the original FM) is available on our homepage[3]. Our implementation[4] converts the SXFM format into FAMA Tool Suite format. Then, we can easily generate a file containing all valid product configurations (operation products in FAMA as defined in [Benavides *et al.*, 2010]).

In our work, we are adding four groups of features to the mined FM in order to organize it and to increase the hierarchy levels of the mined FM. In order to compare the initial FM with the reversed-engineered FM by our approach, we must remove these added groups. The REVPLINE approach produces such this FM without the group of features (*cf.* Figure 6.10).

For correctness, we performed a straightforward test. Using the FAMA tool suite, we computed the list of valid product configurations for cell phone SPL feature model and use it as input to our algorithm (as product configurations in Table 6.1). To identify all valid product configurations from the mined FM, we use the FAMA tool suite to compute the list of valid product configurations. We convert SPLOT FM (SXFM format) to FAMA file format. Then, based on the FAMA operations, we generate a file containing all valid product configurations and the number of products.

We compared the list of product configurations of our reversed-engineered feature model with the list of product configurations originally used as input. Our algorithm produces a set of configurations that includes the set of configurations of the initial FM. We performed an evaluation of the execution time (in ms) of our algorithm using the cell phone FM. The algorithm execution time is equal to 486 *ms*. Table 6.1 shows all valid product configurations for the initial FM. Table 6.2 shows all valid product configurations for the mined FM by our approach (the first 16 product configurations are the same as in Table 6.1).

The initial FM produces 16 valid product configurations. The mined FM produces 31 valid product configurations. We notice that the mined FM contains the 16 product configurations of Table 6.1. The mined FM introduces 15 extra product configurations.

[2]http://gsd.uwaterloo.ca:8088/SPLOT/sxfm.html

[3]http://www.lirmm.fr/CaseStudy

[4]Code : https://code.google.com/p/sxfmtofama/

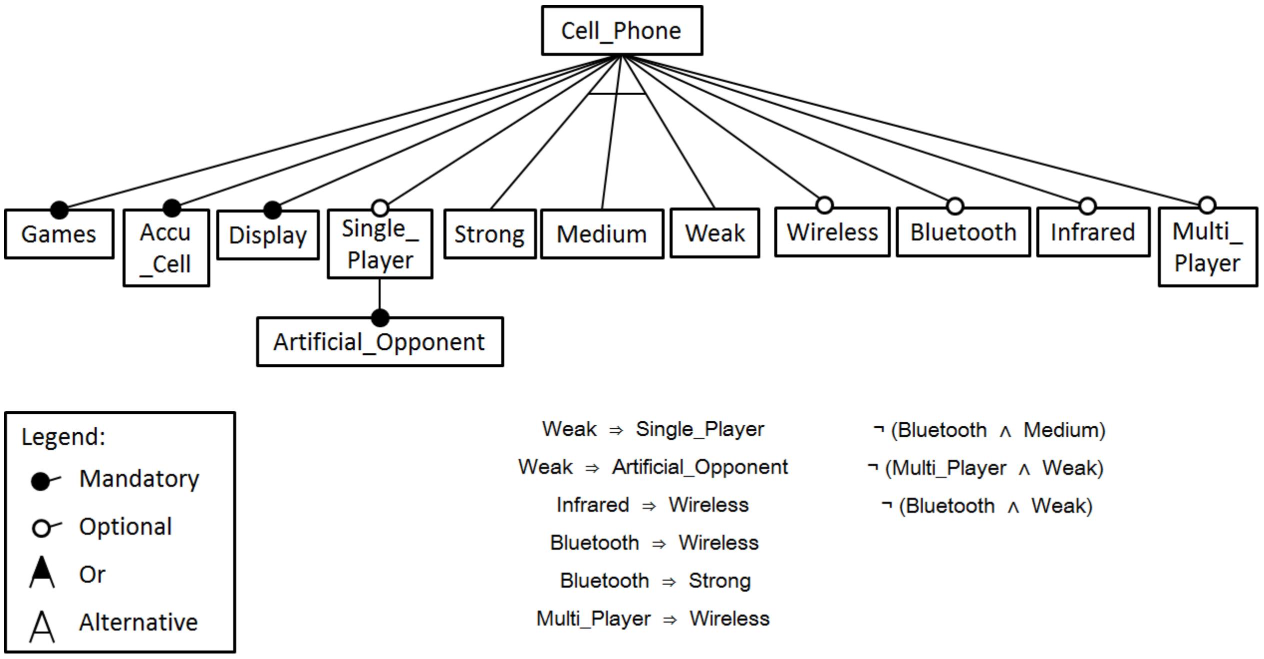


Figure 6.10 : The mined FM without groups of features (*i.e.* without the abstract features).

The mined FM has some extra configurations that correspond to feature selection constraints that have not been detected by our algorithm. 8 of these extra configurations (product-17 to product-24) have "*Games*", but don't have any of its children, which is impossible in the initial feature model (*cf.* Figure 6.2) because alternative edges impose to select at least one child of "Games". 9 configurations (with 2 are common the previous 8), correspond to a same situation with "*wireless*" and its children.

Table 6.3 shows the precision, recall and F-measure metrics that are used to evaluate our results. All measures have values in [0, 1]. If recall equals 1, all relevant product configurations are retrieved. However, some retrieved product configurations might not be relevant. If precision equals 1, all retrieved product configurations are relevant. Nevertheless, relevant product configurations might not be retrieved. If F-Measure equals 1, all relevant product configurations are retrieved. However, some retrieved product configurations might not be relevant. F-Measure defines a trade-off between precision and recall, so that it gives a high value only in cases where both recall and precision are high. The results of this evaluation showed that on the small example, we have good trade-off between precision and recall.

Table 6.2 : All valid product configurations that are defined by the mined FM.

Product configurations	Cell_Phone	Wireless	Infrared	Bluetooth	Accu_Cell	Strong	Medium	Weak	Display	Games	Multi_Player	Single_Player	Artificial_Opponent
Product-1	✗	✗	✗		✗	✗			✗	✗	✗		
Product-2	✗	✗		✗	✗	✗			✗	✗	✗		
Product-3	✗	✗	✗	✗	✗	✗			✗	✗	✗		
Product-4	✗	✗	✗		✗		✗		✗	✗	✗		
Product-5	✗				✗			✗	✗	✗		✗	✗
Product-6	✗				✗	✗			✗	✗		✗	✗
Product-7	✗	✗	✗		✗	✗			✗	✗		✗	✗
Product-8	✗	✗		✗	✗	✗			✗	✗		✗	✗
Product-9	✗	✗	✗	✗	✗	✗			✗	✗		✗	✗
Product-10	✗				✗		✗		✗	✗		✗	✗
Product-11	✗	✗	✗		✗		✗		✗	✗		✗	✗
Product-12	✗	✗	✗		✗			✗	✗	✗		✗	✗
Product-13	✗	✗	✗		✗		✗		✗	✗	✗	✗	✗
Product-14	✗	✗	✗		✗	✗			✗	✗	✗	✗	✗
Product-15	✗	✗		✗	✗	✗			✗	✗	✗	✗	✗
Product-16	✗	✗	✗	✗	✗	✗			✗	✗	✗	✗	✗
Product-17	✗				✗	✗			✗	✗			
Product-18	✗	✗			✗	✗			✗	✗			
Product-19	✗	✗	✗		✗	✗			✗	✗			
Product-20	✗	✗		✗	✗	✗			✗	✗			
Product-21	✗	✗	✗	✗	✗	✗			✗	✗			
Product-22	✗				✗		✗		✗	✗			
Product-23	✗	✗			✗		✗		✗	✗			
Product-24	✗	✗	✗		✗		✗		✗	✗			
Product-25	✗	✗			✗	✗			✗	✗	✗		
Product-26	✗	✗			✗		✗		✗	✗	✗		
Product-27	✗	✗			✗	✗			✗	✗		✗	✗
Product-28	✗	✗			✗		✗		✗	✗		✗	✗
Product-29	✗	✗			✗			✗	✗	✗		✗	✗
Product-30	✗	✗			✗	✗			✗	✗	✗	✗	✗
Product-31	✗	✗			✗		✗		✗	✗	✗	✗	✗

Table 6.3 : The result of the product configurations that are identified by the mined cell phone FM.

	Evaluation Metrics		
	Precision	Recall	F-Measure
Value	0.51	1	0.68

6.5 Conclusion

In this chapter, we proposed an automatic approach to organize the mined documented features into a feature model. Features are organized in a tree which highlights mandatory features, optional features and feature groups (and, or, xor groups). The feature model is completed with requirement and mutual exclusion constraints. We rely on FCA and software configurations to mine a unique and consistent feature model. The FMs are generated in very short time, because our FCA tool scales significantly better than the standard FCA approaches to calculate the lattices.

Thus, we believe that REVPLINE offers a reasonable solution for reverse engineering FMs from software configurations. In this chapter, we present REVPLINE as a new approach to reverse engineering FMs from product-by-feature matrix as we have tried to show throughout the chapter. However, remember the points that make the REVPLINE approach specific:

- REVPLINE accepts as input - to reverse engineering FMs - *product-by-feature matrix* (*i.e.* product configurations). This matrix contains the mined and documented features from source code of software product variants. The product-by-feature matrix can contain any type of software variant artifacts (*i.e.* product descriptions) not limited only to software variant features. The product-by-feature matrix makes our approach more general.
- REVPLINE produces *unique* and *consistent* FMs in very short time. The resulting FM consists of *group of feature constraints* in addition to the *cross-tree constraints* (based on the given software configurations).

Here, we compare the REVPLINE reverse engineering FMs approach to the related work. Table 6.4 shows an overview of the REVPLINE approach where we present the objectives, programmed method, inputs, techniques, outputs, strategies and tool support.

Table 6.4 : Summary of REVPLINE: reverse engineering FMs from software configurations.

Objectives				Programmed method		Inputs		Techniques		Outputs						Strategies	Case study	Tool support
Restructuring variability of SPL	Re-engineering	Domain analysis	Understanding and generative approach	Automatic	Semi-automatic	Software configurations	Product descriptions	Ad hoc algorithm	FCA	Unique FM	Xor-group	And-group	Or-group	Requires constraint	Excludes constraint	Reduce search space		
✘	✘	✘	✘	✘		✘		✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘

Part III

Experimentation

CHAPTER 7

EXPERIMENTATION

If A is success in life, then A = x + y + z. Work is x; y is play; and z is keeping your mouth shut.
Albert EINSTEIN

Preamble

This chapter shows the experiments that we run to validate our proposal. Section 7.1 gives an introduction of this chapter. Section 7.2 presents the ArgoUML-SPL case study and validation part. Section 7.3 presents Health complaint-SPL case study and validation part. Section 7.4 presents Mobile Media product variants case study and validation part. Section 7.5 gives the obtained FMs by applying our approach on sets of valid feature combinations. Finally Section 7.6 concludes this chapter.

7.1 Introduction

In this chapter, we present the experiments that we conducted to validate the REVPLINE approach. The results of the application of our approach over a set of software where each is implemented as a set of variants allowed us to assess its merits and limitations. We present each experiment as two parts: the case study part, where we explain and describe the selected case study; and the validation part, where we show and discuss the obtained results. We apply our approach on three real case studies: ❶ *ArgoUML-SPL*, ❷ *Health complaint-SPL* and ❸ *Mobile Media*. ArgoUML-SPL and Health complaint-SPL represent real SPLs. Mobile Media represents real software product variants. The three case studies are Java software systems. We executed all experiments on a Windows 7 system, running at 2.30 GHz, and with RAM of 8 GB.

The advantage of having *three* case studies is that they implement *variability* at different levels (package, class, attribute, method or method body). Variability in Mobile Media variants is present at method and method body levels. Variability in ArgoUML-SPL variants (and Health complaint-SPL variants) is present at package and class levels. In addition, ArgoUML-SPL, Health complaint-SPL and Mobile Media software product variants are *well documented* and their features, use-case diagrams and feature models are available for comparison to our results and validation of our approach. We used 10 software systems for ArgoUML-SPL, 10 for Health complaint-SPL and 4 software variants for Mobile Media software product variants. The selected software (*i.e.* number of variants) is covering all known features of these variants.

The three case studies show *different sizes*: ArgoUML-SPL (large systems), Health complaint-SPL (medium systems) and Mobile Media (small systems). The different complexity levels show the scalability of our approach to dealing with such systems. The selected case studies are used to assess many approaches in the field of our study.

In order to evaluate our approach and base on our knowledge about case studies (software feature implementations, feature names, use-case diagrams and FM), we have used three measures: *precision*, *recall* and *F-Measure* to assess the obtained results. The effectiveness of IR methods is measured by these metrics. We rely on these metrics to show the efficiency of our approach. For each case study, we validate the result of the feature mining, feature documentation and reverse engineering FM.

In this chapter, we present each experiment in a separate section. Each section consists of two parts: the case study part and the validation part. We should mention here that the validation part contains three parts: feature location, feature documentation and FM reverse engineering.

7.2 ArgoUML-SPL Case Study

7.2.1 ArgoUML-SPL Description

ArgoUML[1] is a Java-based, open source software, widely used for designing systems in UML. ArgoUML is the leading open source UML modeling tool and includes support for all standard UML 1.4 diagrams. It runs on any Java platform. ArgoUML-SPL is described in [Couto *et al.*, 2011].

We selected 10 products of ArgoUML-SPL. Naturally, the selected products cover all features

[1]ArgoUML-SPL source code : `http://argouml-spl.tigris.org/source/browse/argouml-spl/`

of ArgoUML-SPL. ArgoUML[2] variants are presented in Table 7.1 characterized by metrics LoC (Lines of Code), NoP (Number of Packages), NoC (Number of Classes) and number of OBEs.

Table 7.1 : ArgoUML software product variants.

Product #	ArgoUML Product Description	LoC	NoP	NoC	Number of OBEs
P1	All optional features disabled	82,924	55	1,243	74,444
P2	All optional features enabled	120,348	81	1,666	100,420
P3	Only logging feature disabled	118,189	81	1,666	98,988
P4	Only cognitive feature disabled	104,029	73	1,451	89,273
P5	Only sequence diagram disabled	114,969	77	1,608	96,492
P6	Only use-case diagram disabled	117,636	78	1,625	98,468
P7	Only deployment diagram disabled	117,201	79	1,633	98,323
P8	Only collaboration diagram disabled	118,769	79	1,647	99,358
P9	Only state diagram disabled	116,431	81	1,631	97,760
P10	Only activity diagram disabled	118,066	79	1,648	98,777

The FM for the ArgoUML-SPL as defined by its designers is presented in [Couto *et al.*, 2011]. It contains nine features. Figure 7.1 presents the feature model of the ArgoUML-SPL. For creating ArgoUML-SPL, nine features representing functional and non-functional requirements have been selected. The first feature (logging) has been selected as a representative of a non-functional requirement. The other eight features represent functional concerns: class diagram, activity diagram, state diagram, collaboration diagram, sequence diagram, use-case diagram, deployment diagram and cognitive support.

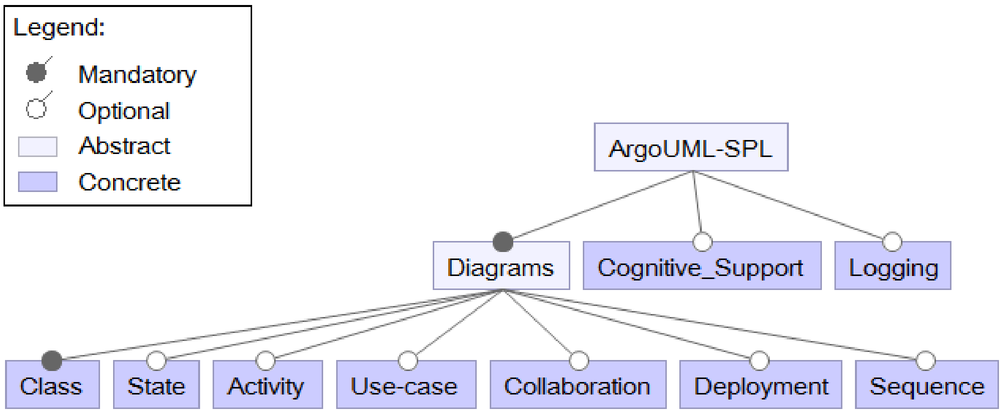


Figure 7.1 : ArgoUML-SPL feature model [Couto *et al.*, 2011].

Figure 7.2 shows the use-case diagram of the second release of ArgoUML. The description of each use-case is available at ArgoUML website[3].

[2]ArgoUML-SPL : http://argouml-spl.tigris.org/
[3]http://argouml-spl.tigris.org/

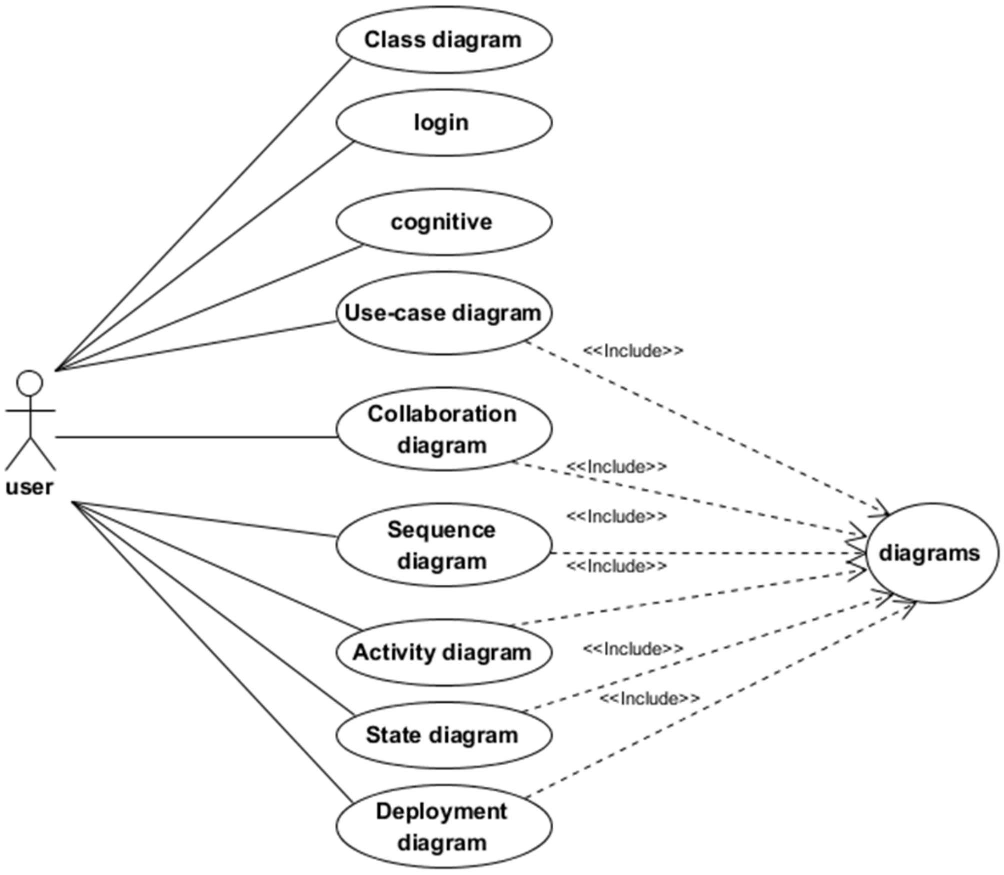


Figure 7.2 : The use-case diagram of the second release of ArgoUML software variants.

7.2.2 ArgoUML Validation

7.2.2.1 Feature Location

❶ Feature location based on lexical similarity: Here, we show and discuss the obtained results for feature location in a collection of ArgoUML software variants. The experiment shows us that all ArgoUML-SPL features are implemented at package and class levels except the logging feature which is implemented at the method body level (*i.e.* method invocation and attribute access).

The obtained result for feature location in ArgoUML software variants is presented in Table 7.2. For readability's sake in Table 7.2, we manually associated feature names to atomic blocks (*i.e.* the mined feature implementations), based on the study of the content of each block and on our knowledge on software. Of course, this does not impact the quality of our results.

Results show that *precision* appears to be high for all optional features. This means that all mined OBEs grouped as features are relevant. This result is due to search space reduction. In most cases, each block of variation (BV) corresponds to one and only one feature. For mandatory features, precision is also quite high thanks to our clustering technique that identifies atomic blocks (*i.e.* feature implementations) based on FCA and LSI. However, precision is smaller than the one obtained for optional features. This deterioration can be explained by the fact that we do not perform search space reduction for the common block (CB).

Considering the *recall* metric, its average value is 67% for ArgoUML. This means that most

OBEs that compose features are mined. We have manually identified OBEs which should have been mined and were not. We found that these non-mined OBEs used different vocabularies from mined OBEs'. This is a known limitation of LSI which is based on lexical similarity. Considering the *F-Measure* metric, our approach has values that range from 63% to 88%. This means that most OBEs that compose features are mined and shows the efficiency of our approach.

The most important parameter to LSI is the number of term-topics (*i.e.* k-Topics). A term-topic is a collection of terms that co-occur frequently in the documents of the corpus. We need enough term-topics to capture real term relations. In our work, we cannot use a fixed number of topics for LSI because we have *blocks of variation* (*i.e.* clusters) with different sizes.

Table 7.2 : Feature location in a collection of ArgoUML software variants based on lexical similarity.

#	Feature Name	Feature Type		k-Topics	Evaluation Metrics		
		Mandatory	Optional		Precision	Recall	F-Measure
1	Class diagram	✘		0.03	72%	56%	63%
2	Diagrams		✘	0.06	100%	80%	88%
3	Deployment diagram		✘	0.05	100%	74%	85%
4	Collaboration diagram		✘	0.06	100%	67%	80%
5	Use-case diagram		✘	0.03	100%	64%	78%
6	State diagram		✘	0.03	100%	69%	81%
7	Sequence diagram		✘	0.02	100%	67%	80%
8	Activity diagram		✘	0.06	100%	63%	77%
9	Cognitive support		✘	0.01	100%	70%	82%
10	Logging		✘	0.02	100%	60%	75%

Table 7.3 : Size metric and the variability levels of ArgoUML features.

#	Feature Name	LoC	Variability levels				
			Package	Class	Attribute	Method	Method Body
1	Cognitive support	16,319	✗	✗			
2	Activity diagram	2,282	✗	✗			
3	State diagram	3,917	✗	✗			
4	Collaboration diagram	1,579	✗	✗			
5	Sequence diagram	5,379	✗	✗			
6	Use-case diagram	2,712	✗	✗			
7	Deployment diagram	3,147	✗	✗			
8	Logging	2,159					✗

Table 7.3 shows the size metric and the variability levels of ArgoUML features. Listing 7.1 shows part of the sequence diagram feature implementation.

As mentioned in the feature location chapter (*cf.* Chapter 4), our approach identifies a junction as a feature. Table 7.4 represents the mined junctions from ArgoUML software variants. We know that these BVs represent junctions based on our knowledge about this software family. The column (# OBEs) in Table 7.4 represents the number of OBEs that represent each junction.

❷ Feature location based on lexical similarity and structural dependency: Features of ArgoUML-SPL are implemented at the *package* or *class* level except the *logging* feature (at method body level); as a consequence, ArgoUML-SPL is appropriate for application of both the lexical and

```
Package (argouml.uml.diagram.sequence.ui)
Class (ModeChangeHeight_argouml.uml.diagram.sequence.ui)
Attribute (serialVersionUID_SequenceDiagramGraphModel)
Method (initialize()_PropPanelActionSequence)
Local Variable (lay_UMLSequenceDiagram())
Class (ActionSetOperation_sequence2.diagram)
Method Invocation (info ["SequenceDiagram Module enabled."]_enable())
Method (relocate(base)_UMLSequenceDiagram)
Method Invocation (debug ["Created sequence diagram"]_UMLSequenceDiagram())
                                         .
                                         .
                                         .
```

LISTING 7.1 : Part of the sequence diagram feature implementation.

Table 7.4 : The identified junctions between feature implementations of ArgoUML-SPL.

#	Junction	# OBEs
1	Junction between *cognitive* and *deployment* features	745
2	Junction between *cognitive* and *sequence* features	55
3	Junction between *sequence* and *collaboration* features	111
4	Junction between *state* and *logging* features	6
5	Junction between *deployment* and *logging* features	18
6	Junction between *collaboration* and *logging* features	13
7	Junction between *use-case* and *logging* features	22
8	Junction between *sequence* and *logging* features	51
9	Junction between *activity* and *logging* features	3
10	Junction between *cognitive* and *logging* features	169
11	Junction between *activity* and *state* features	57

structural similarity measures. Table 7.5 summarizes the obtained results. For readability's sake, we manually associated feature names to feature implementations.

Table 7.5 : Feature location in a collection of ArgoUML software variants based on lexical and structural similarity.

#	Feature Name	Feature Type		NoC	# of couplings	k-Topics	Evaluation Metrics		
		Mandatory	Optional				Precision	Recall	F-Measure
1	Class diagram	✗		55	170	0.03	72%	100%	82%
2	Diagrams		✗	18	16	0.06	100%	100%	100%
3	Sequence diagram		✗	57	140	0.02	100%	100%	100%
4	Deployment diagram		✗	20	8	0.05	100%	100%	100%
5	Collaboration diagram		✗	19	2	0.06	100%	100%	100%
6	Cognitive support		✗	207	6556	0.01	100%	100%	100%
7	Use-case diagram		✗	39	14	0.03	100%	100%	100%
8	State diagram		✗	35	48	0.03	100%	100%	100%
9	Activity diagram		✗	18	8	0.06	100%	100%	100%

Results show that precision seems to be high for all optional features. This means that all mined OBEs gathered as features are relevant. This result is due to search space reduction. In most cases, each BV corresponds to a single feature. For mandatory features, precision is also quite high thanks to our clustering technique that identifies atomic blocks based on lexical and structural similarity. However, precision is smaller than the one obtained for optional features. This deterioration can again be explained by the fact that we do not perform search space reduction for the CB.

Considering the recall metric, its average value is 100% for ArgoUML-SPL. This means that all OBEs that compose features are mined. Considering the F-Measure metric, our approach has the value of 100%. This means that all OBEs that compose features are mined and provides initial evidence with regard to the efficiency of our approach.

All common and optional features are mined for ArgoUML-SPL except the *logging* feature; the reason behind this limitation is that the logging feature is implemented in a method body and our approach only considers the software variants which variability is represented mainly in the package or class level. The results of this evaluation showed that most of the features were identified and proves the scalability of our feature mining approach.

We compare the two approaches (lexical similarity versus lexical similarity and structural dependency) and the result is presented in Table 7.6.

Table 7.6 : Comparing the two ways: lexical versus structural and lexical similarity.

ArgoUML-SPL	Precision	Recall	F-Measure	Number of Junctions
Lexical similarity	97%	67%	79%	11
Lexical and structural similarity	97%	100%	98%	1

From Table 7.6, we can note that the approach using lexical and structural similarity gives better results than the lexical approach alone. For the precision value, it is the same for both approaches. The recall and F-Measure metric values are better than lexical approach alone.

The reason behind this result is the use of the structural dependency information contained in source code to increase the precision and recall of an LSI method. In other words, the dependency information helps to get additional correct links between OBEs.

Through Table 7.6 we can observe that the number of junctions resulting from the use of the lexical approach are bigger than in the combined approach. In the combined approach, there is one junction between cognitive support and deployment diagram. This junction consists of 13 classes.

7.2.2.2 Feature Documentation

The feature documentation approach accepts as inputs: ❶ the mined feature implementations and ❷ the use-case diagrams (in addition to the textual descriptions, *cf.* Listing 7.3) of the software variants. As we presented in chapter 5, we exploit commonality and variability across software variants at feature implementations and use-cases levels in order to apply LSI in an efficient way.

For this case study, we rely on the same product descriptions of this family as presented in Table 7.1, we get a set of hybrid blocks which all contain one and only one feature implementation and use-case. The values of *precision*, *recall* and *F-measure* metrics are exactly 100%. Reducing the search space for use-cases and feature implementations across software variants is certainly the reason of this performance and results. The description of each product specifies the disabled or the enabled features. Basically speaking, products P1 and P2 represent respectively the core product (all optional features disabled) and the full product (all optional features enabled). The other products represent cases in which only one of the features has been disabled.

In this section, we rely on new product configurations (*i.e.* descriptions), as presented in Table 7.7, to get hybrid blocks with more than one feature implementation and use-case. In order to document the mined feature implementations from the ArgoUML case study, we rely on the use-case diagrams of these software variants.

Table 7.7 : The descriptions of the ArgoUML software product variants.

	Class	Diagrams	Cognitive support	Logging	State	Use-case	Deployment	Collaboration	Sequence	Activity
Product-1	✓									
Product-2	✓	✓								
Product-3	✓	✓	✓	✓						
Product-4	✓	✓					✓			
Product-5	✓	✓						✓	✓	
Product-6	✓	✓			✓					
Product-7	✓	✓				✓				✓

Table 7.8 summarizes the obtained results (*i.e.* feature names) of these variants. The REVPLINE system documents the mined feature implementation by assigning name and description (in the form of sentences) based on use-case name and textual description of use-case. For example, in the FM of ArgoUML-SPL there are two features called: use-case and collaboration. The proposed names for these features by our approach are: use-case diagram and collaboration diagram. The examples below show the names and descriptions of these features.

Example 7.1. *Use-case diagram: "a use-case is a set of scenarios that describes an interaction between a user and a system. A use-case diagram displays the relationship among actors and use-cases. The two main components of a use-case diagram are use-cases and actors".*

Example 7.2. *Collaboration diagram: "collaboration diagram is used to show how objects interact to perform the behavior of a particular use-case, or a part of a use-case. Along with sequence diagram, collaboration diagram is used by designers to define and clarify the roles of the objects that perform a particular flow of events of a use-case. Collaboration diagram is the primary source of information used to determining class responsibilities and interfaces".*

From Table 7.8, we observe that the *recall* values are 100% of all the features that are documented. The recall values are an indicator for the efficiency of our approach. The values of precision are between [50% − 100%], which is high. F-Measure values rely on precision and recall values. The values of F-Measure are high too, between [66% − 100%] for the documented features.

Reducing the search space for use-cases and feature implementations across software variants is certainly the reason of this performance. In most cases, the contents of hybrid blocks are in the range of [1−2] use-cases and feature implementations. Another reason for this good result is that a common vocabulary is used in the use-case descriptions and feature implementations, thus lexical similarity was a suitable tool.

Table 7.8 : Features documented from ArgoUML software variants based on use-case diagrams.

#	Feature Name	Hybrid block #	k-Topics	Evaluation Metrics		
				Recall	Precision	F-Measure
1	Class diagram	HB−1	1	100%	100%	100%
2	Diagrams	HB−2	1	100%	100%	100%
3	Logging	HB−3	2	100%	50%	66%
4	Cognitive support	HB−3	2	100%	100%	100%
5	Deployment diagram	HB−4	1	100%	100%	100%
6	Collaboration diagram	HB−5	2	100%	50%	66%
7	Sequence diagram	HB−5	2	100%	50%	66%
8	State diagram	HB−6	1	100%	100%	100%
9	Activity diagram	HB−7	2	100%	100%	100%
10	Use-case diagram	HB−7	2	100%	100%	100%

The column (*k-Topics*) in Table 7.8 represents *the number of term-topics*. In our work, the number of k-Topics is equal to the number of feature implementations for each corpus (hybrid block). All feature names produced by our approach, in the column (*Feature Name*) of Table 7.8, represent the names of the use-cases.

7.2.2.3 FM Reverse Engineering

Table 7.9 shows the product-by-feature matrix for ArgoUML software variants. The technique used to identify the FM relies on FCA. First, a formal context, where *objects* are product variants and *attributes* are features (*cf.* Table 7.9), is defined. The corresponding AOC-poset is then calculated (*cf.* Figure 7.3).

Table 7.9 : The Product-by-feature matrix for ArgoUML software product variants.

	ArgoUML-SPL	Class diagram	Diagrams	Use-case diagram	Collaboration diagram	Cognitive support	Activity diagram	Deployment diagram	Sequence diagram	State diagram	Logging
Product−1	✖	✗									
Product−2	✖	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Product−3	✖	✗	✓	✓	✓	✓	✓	✓	✓	✓	
Product−4	✖	✗	✓	✓	✓		✓	✓	✓	✓	✓
Product−5	✖	✗	✓	✓	✓	✓	✓	✓		✓	✓
Product−6	✖	✗	✓		✓	✓	✓	✓	✓	✓	✓
Product−7	✖	✗	✓	✓	✓	✓	✓		✓	✓	✓
Product−8	✖	✗	✓	✓		✓	✓	✓	✓	✓	✓
Product−9	✖	✗	✓	✓	✓	✓	✓	✓	✓		✓
Product−10	✖	✗	✓	✓	✓	✓		✓	✓	✓	✓

Product-by-feature matrix
(✖ the root feature; ✗ mandatory feature; ✓ optional feature)

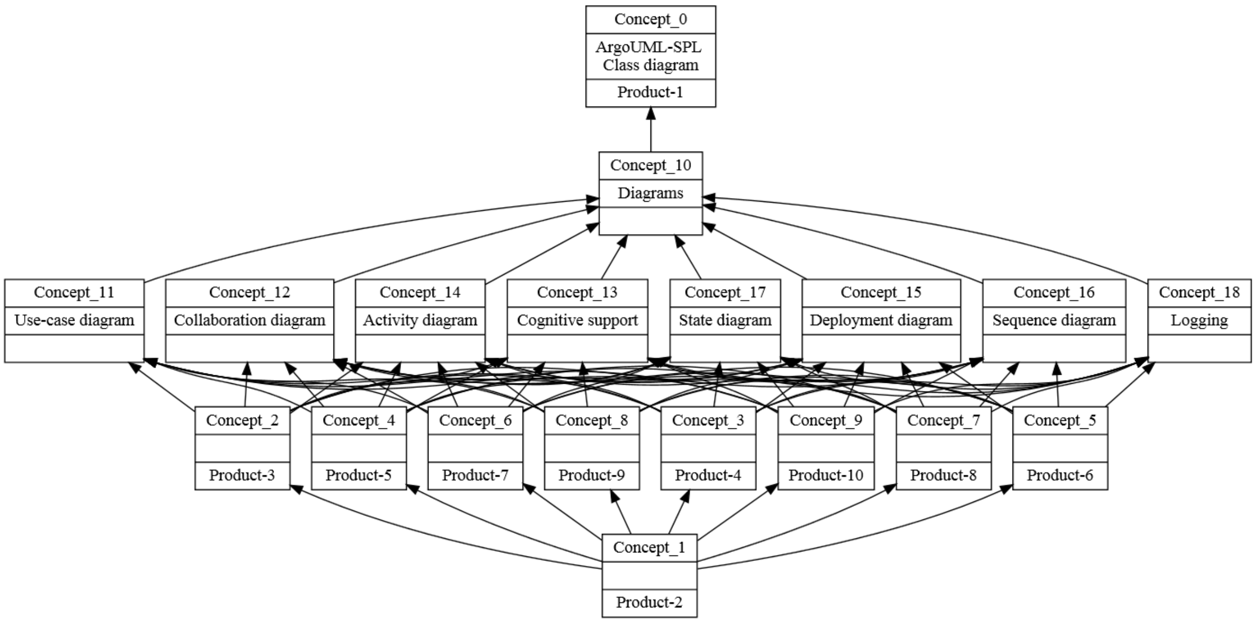


Figure 7.3 : The AOC-poset for the formal context of Table 7.9.

Based on the AOC-poset in Figure 7.3, we identify the FM which represents the mined and documented features from ArgoUML. Figure 7.4 shows the mined FM for ArgoUML-SPL. The resulting FM describes exactly the given product configurations. The needed time to identify the FM is equal to 492 ms. The time needed to extract the FM includes all processes (starting from converting the product configurations to formal context and generating the AOC-poset, up to analyzing the AOC-poset and generating the FM).

We should mention that the identified FM consists only of *concrete* features (without *abstract* features). The abstract feature corresponds to a group of features or to the root feature in the FM. The abstract feature cannot be identified by our approach in general. The feature names in the mined FM are very close to the feature names in the original FM. The extracted and original FM both consist of three levels of hierarchy.

The extracted FM is containing a mandatory feature, nine optional features and eight constraints of type requires. We can note that the identified FM by our approach introduces a unique and consistent FM. Moreover, the mined FM is introducing new constraints which do not exist in the original FM (the manually designed FM by the authors of ArgoUML). The results of this evaluation provide evidence with regard to the efficiency of our approach and show the scalability of our FM reverse engineering approach.

Results show that precision appears to be not very high for ArgoUML-SPL (*i.e.* 60%). This means that many of the identified product configurations of the mined FM are extra configurations (not in initial set that is defined by the original FM). Considering the recall metric, it is value is 100% for ArgoUML-SPL. This means that product configurations defined by the initial FM are included in the product configurations derived from the mined FM. Considering the F-Measure metric, our approach has value 75%.

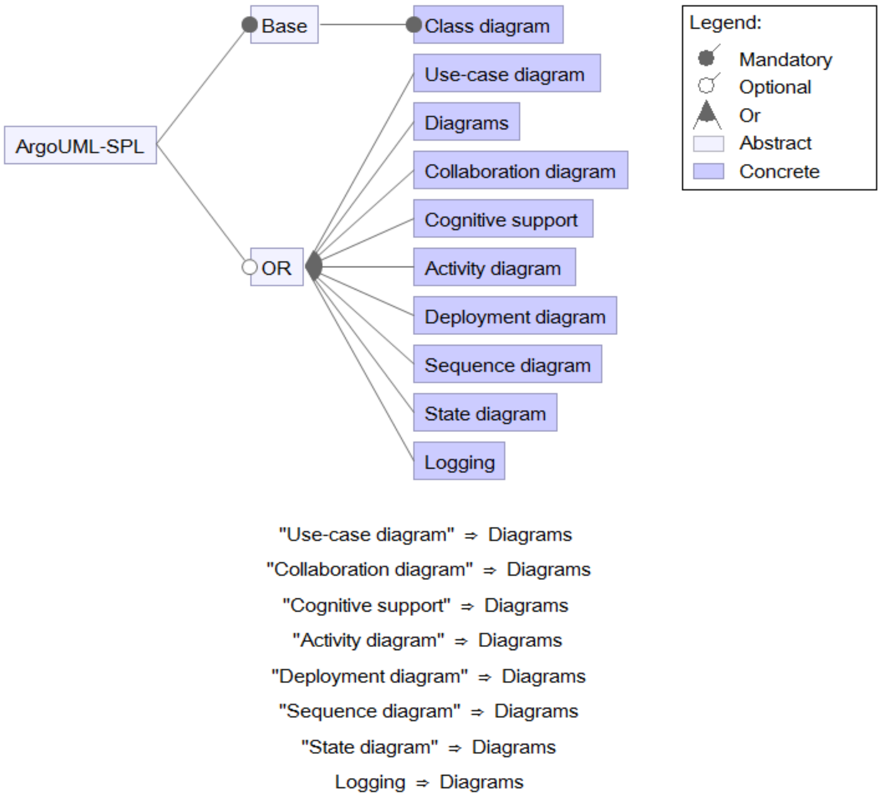


Figure 7.4 : The extracted FM from Table 7.9.

7.3 Health complaint-SPL Case Study

7.3.1 Health complaint-SPL Description

The public Health complaint[4] is a SPL based on three *legacy applications* that belong to the domain of *public health complaints*. Health complaint is a Java-based open source web application that manages health related records and complaints. The system was initially developed in 2001 and has undergone 9 releases to add new features and fix previous bugs. These legacy applications allow citizens to report complaints via Internet. The types of complaints encompass *food*, *animal* and *special* complaints.

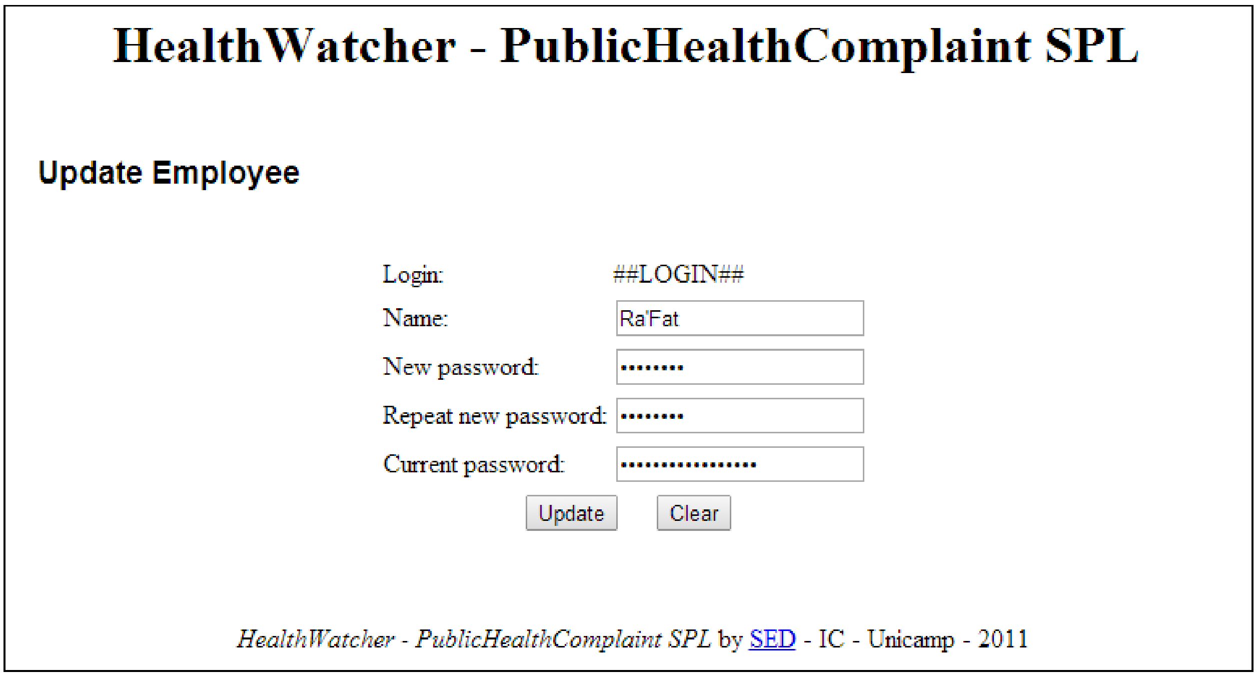
Health complaint-SPL[5] variants are presented in Table 7.10. We select 10 products of health complaint as published in [Tizzei *et al.*, 2012]. Naturally, the selected products cover all features of health complaint software variants. These software variants include 10 features. Health complaint software variants are presented in Table 7.10 characterized by metrics LoC, NoP, NoC and number of OBEs.

Each new release of the system *adds new features* or *fixes previous bugs*. For instance, release 1 contains the *core* system (the mandatory features) whilst release 2 represents the core system with the functionalities of *complaint update*, *register new employee*, *update employee* and *query information*. The different features of the system and the releases where they were included are described in Table 7.14. Figure 7.5 shows the "update employee" feature of the public health complaint application.

[4]http://www.ic.unicamp.br/~tizzei/phc/jss2013/index.html
[5]Source code : https://github.com/leotizzei/PublicHealthComplaintAO_01

Table 7.10 : Health complaint software product variants.

Product #	Health complaint Product Description	LOC	NOP	NOC	Number of OBEs
P1	Base - no extensions applied	5,288	22	88	6,603
P2	Command pattern applied	5,646	23	92	6,867
P3	State pattern applied	6,112	24	104	7,407
P4	Observer pattern applied	6,222	26	106	7,536
P5	Adapter pattern applied v1	6,379	26	108	7,631
P6	Abstract factory pattern applied v1	6,417	27	112	7,659
P7	Adapter pattern applied v2	6,441	27	116	7,648
P8	Abstract factory pattern applied v2	6,468	28	120	7,669
P9	Evolution - New functionality added	7,709	28	132	9,079
P10	Exception handling applied	7,591	29	135	9,084


HealthWatcher - PublicHealthComplaint SPL

Update Employee

Login:	##LOGIN##
Name:	Ra'Fat
New password:	••••••••
Repeat new password:	••••••••
Current password:	•••••••••••••••••

HealthWatcher - PublicHealthComplaint SPL by SED - IC - Unicamp - 2011

Figure 7.5 : Public health complaint application interface: update employee screen.

Release 1 supports core health complaint features: special complaint, food complaint, animal complaint, exception handling and persistence. Release 2 supports complaint update, register new employee, update employee and query information optional features, together with the core ones. Release 3 has the core health complaint features and a new optional feature called receive alerts via feeds. Release 4 has the core health complaint features and a new optional feature called complaint status. Release 9 has the core health complaint features and a new optional feature called update medical speciality.

Release 5 (*resp.* release 10) supports core health complaint features, together with the optional features of previous releases. This release does not support new features and fixes previous bugs. It contains only 3 (*resp.* 5) additional classes compared to the Release 4 (*resp.* release 9). Release 6 does not support new features and fixes previous bugs. It contains only 4 additional classes compared to the release 5 to fix some bugs. Release 7 (*resp.* release 8) doesn't support new features and fixes previous bugs. It contains only 4 additional classes compared to release

6 (*resp.* release 7) (*cf.* Figure 7.6).

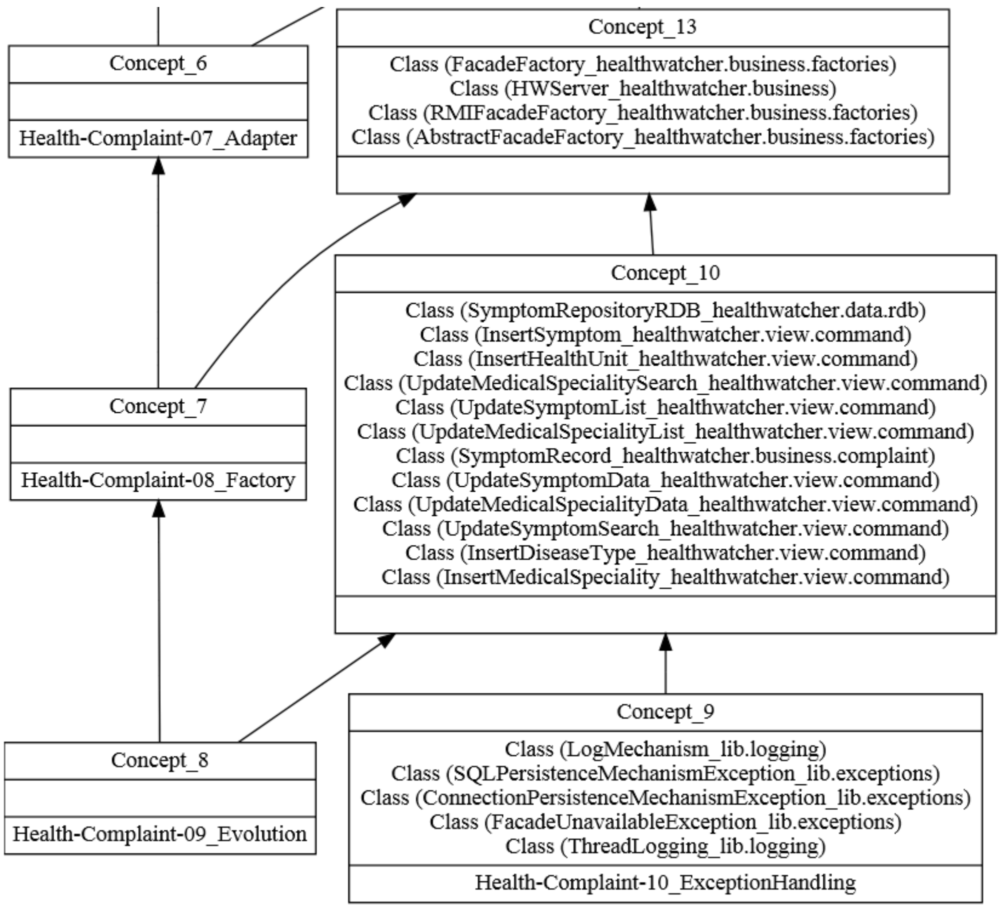

Concept_6
Health-Complaint-07_Adapter
Concept_13
Class (FacadeFactory_healthwatcher.business.factories)
Class (HWServer_healthwatcher.business)
Class (RMIFacadeFactory_healthwatcher.business.factories)
Class (AbstractFacadeFactory_healthwatcher.business.factories)
Concept_10
Class (SymptomRepositoryRDB_healthwatcher.data.rdb)
Class (InsertSymptom_healthwatcher.view.command)
Class (InsertHealthUnit_healthwatcher.view.command)
Class (UpdateMedicalSpecialitySearch_healthwatcher.view.command)
Class (UpdateSymptomList_healthwatcher.view.command)
Class (UpdateMedicalSpecialityList_healthwatcher.view.command)
Class (SymptomRecord_healthwatcher.business.complaint)
Class (UpdateSymptomData_healthwatcher.view.command)
Class (UpdateMedicalSpecialityData_healthwatcher.view.command)
Class (UpdateSymptomSearch_healthwatcher.view.command)
Class (InsertDiseaseType_healthwatcher.view.command)
Class (InsertMedicalSpeciality_healthwatcher.view.command)
Concept_7
Health-Complaint-08_Factory
Concept_8
Health-Complaint-09_Evolution
Concept_9
Class (LogMechanism_lib.logging)
Class (SQLPersistenceMechanismException_lib.exceptions)
Class (ConnectionPersistenceMechanismException_lib.exceptions)
Class (FacadeUnavailableException_lib.exceptions)
Class (ThreadLogging_lib.logging)
Health-Complaint-10_ExceptionHandling

Figure 7.6 : Part of the AOC-poset of Health complaint-SPL.

The FM for the Health complaint-SPL as manually designed by the authors of Health complaint-SPL is presented in [Tizzei *et al.*, 2012]. Health complaint-SPL FM contains 10 features. Figure 7.7 shows the FM of the Health complaint-SPL. This FM consists of: 4 groups of features from abstract type (complaint management, complaint specification, support services and infrastructure services), 5 mandatory features (animal complaint, food complaint, special complaint, exception handling and persistence), 5 optional features (complaint update, RSS feeds, query information, register new employee and update employee), without cross-tree constraints (we can call it basic FM) and the root feature "public health complaint system SPL".

Figure 7.8 shows the use-case diagram for the public health complaint SPL[6]. There is a symbiotic relationship between features and use-cases. Feature models focus on specifying the features variability by means of a graphical user-friendly and hierarchical structure. Furthermore, use-cases specify the interaction between user and system, and also the system behavior [Gomaa, 2004]. Exception handling and persistence are non-functional requirements and represent non-functional use-cases[7].

[6] http://www.ic.unicamp.br/~tizzei/phc/jss2013/node17.html
[7] http://www.ic.unicamp.br/~tizzei/phc/jss2013/node18.html

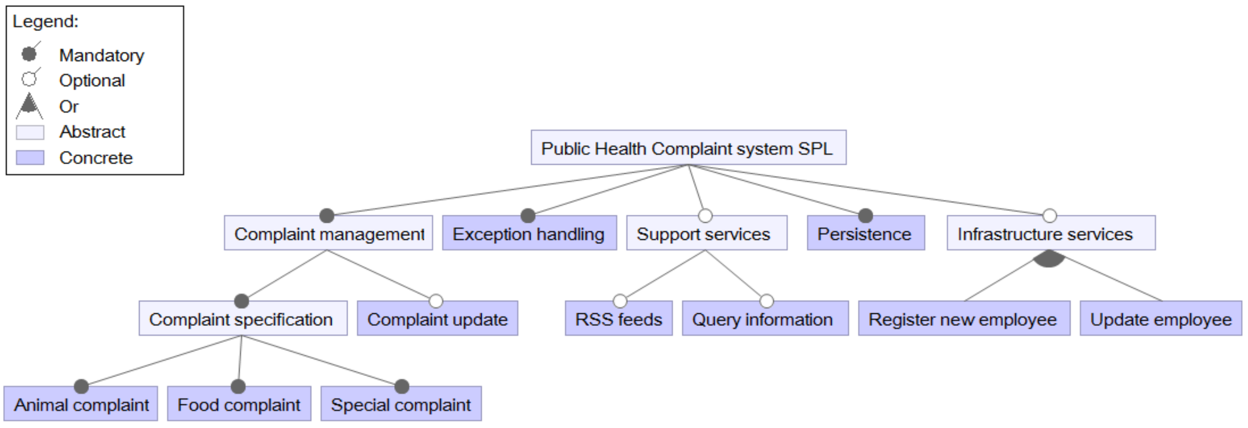


Figure 7.7 : Feature model for the public Health complaint-SPL.

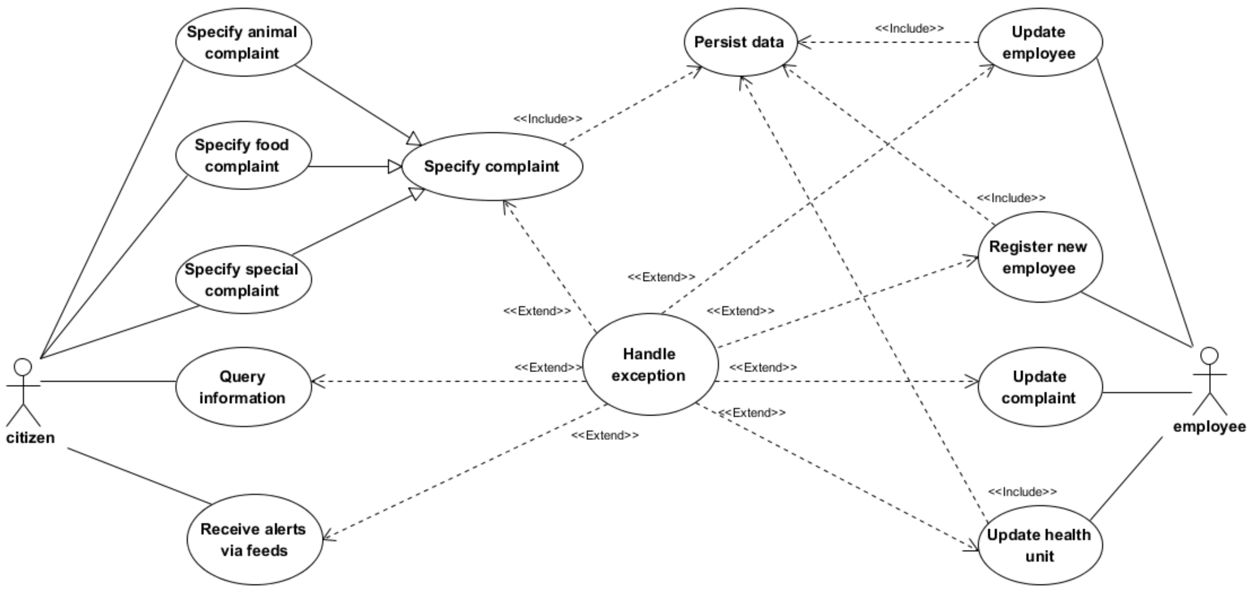


Figure 7.8 : Public Health complaint-SPL use case diagram.

7.3.2 Health Complaint Validation

7.3.2.1 Feature Location

To evaluate the feasibility of our approach to locating the source code elements that implement a specific feature, we conducted an experiment on *Health complaint-SPL*. Table 7.11 summarizes the obtained results for feature location in a collection of health complaint software variants. For readability's sake, we manually associated feature names to the mined feature implementations. For health complaint software variants, we rely on the *lexical and structural similarity* to mine their features. The reason behind this choice is that variability across these variants appears at *class level*.

Results show that the *precision* metric appears high, its average value is 75% for health complaint product variants. This means that most of the mined OBEs grouped as features are relevant. This result is due to search space reduction. In most cases, each block of variation (BV) corresponds to one and only one feature. For mandatory features, precision is also quite high

Table 7.11 : Features mined from health complaint software variants.

#	Feature Name	Feature Type		k-Topics	Evaluation Metrics		
		Mandatory	Optional		Precision	Recall	F-Measure
1	Special complaint	✗		0.10	83%	100%	90%
2	Food complaint	✗		0.10	83%	100%	90%
3	Animal complaint	✗		0.10	83%	100%	90%
4	Update health unit	✗		0.10	60%	100%	75%
5	Exception handling	✗		0.10	62%	100%	76%
6	Persistence	✗		0.10	57%	100%	72%
7	Complaint update		✗	0.20	81%	100%	90%
8	Register new employee		✗	0.20	60%	100%	75%
9	Update employee		✗	0.20	50%	100%	66%
10	Query information		✗	0.20	56%	100%	72%
11	Receive alerts via feeds		✗	0.06	100%	100%	100%
12	Complaint status		✗	0.04	100%	100%	100%
13	Update medical speciality		✗	0.01	100%	100%	100%

thanks to our clustering technique that identifies feature implementations based on *lexical* and *structural* similarity.

Considering the *recall* metric, its average value is 100% for health complaint product variants. This means that all OBEs that compose features are mined. Considering the *F-Measure* metric, our approach has values that range from 66% to 100%. F-Measure average is equal to 84%. This means that most OBEs that compose features are mined. F-Measure defines a trade-off between precision and recall, so that it gives a high value only in cases where both recall and precision are high.

As we said before, the most important parameter to LSI is the number of term-topics (*i.e.* k-Topics). In our work we cannot use a fixed number of topics for LSI because we have clusters with different sizes. Results also show that we obtain three features not included in the FM of Health complaint-SPL. These features are *update health unit*, *complaint status* and *update medical speciality*. For these mined feature implementations, two of them (complaint status and update medical speciality) are documented based on the OBE names (*cf.* Table 7.12). The update health unit feature is documented based on the use-case diagram (there is use-case called update health unit). Each feature implementation is mined as a collection of OBEs. Listing 7.2 shows the *complaint status* feature implementation. Each feature implementation consists of a set of OBEs (*i.e.* set of classes).

```
Class (FoodComplaintStateClosed_healthwatcher.model.complaint.state)
Class (ComplaintStateClosed_healthwatcher.model.complaint.state)
Class (FoodComplaintState_healthwatcher.model.complaint.state)
Class (FoodComplaintStateOpen_healthwatcher.model.complaint.state)
Class (SpecialComplaintStateOpen_healthwatcher.model.complaint.state)
Class (ComplaintState_healthwatcher.model.complaint.state)
Class (SpecialComplaintStateClosed_healthwatcher.model.complaint.state)
Class (AnimalComplaintState_healthwatcher.model.complaint.state)
Class (AnimalComplaintStateClosed_healthwatcher.model.complaint.state)
Class (ComplaintStateOpen_healthwatcher.model.complaint.state)
Class (SpecialComplaintState_healthwatcher.model.complaint.state)
Class (AnimalComplaintStateOpen_healthwatcher.model.complaint.state)
```

LISTING 7.2 : The *complaint status* feature implementation.

7.3.2.2 Feature Documentation

In order to document the mined feature implementations from health complaint case study, we rely on the use-case diagrams of these software variants. Table 7.12 summarizes the obtained results (*i.e.* feature names) for this case study.

Table 7.12 : Features documented from Health complaint software product variants.

#	Feature Name	Hybrid block #	k-Topics	Evaluation Metrics		
				Recall	Precision	F-Measure
1	Specify animal complaint	HB−1	6	100%	50%	66%
2	Specify food complaint	HB−1	6	100%	50%	66%
3	Specify special complaint	HB−1	6	100%	50%	66%
4	Update health unit	HB−1	6	100%	100%	100%
5	Handle exception	HB−1	6	100%	100%	100%
6	Persist data	HB−1	6	100%	100%	100%
7	Update employee	HB−2	4	100%	50%	66%
8	Register new employee	HB−2	4	100%	50%	66%
9	Update complaint	HB−2	4	100%	100%	100%
10	Query information	HB−2	4	100%	100%	100%
11	Receive alerts via feeds	HB−3	3	100%	100%	100%
12	Complaint status	HB−3	3	–	–	–
13	Update medical speciality	HB−3	3	–	–	–

In the FM of Health complaint-SPL [Tizzei *et al.*, 2012] there is a feature called *food complaint*. The name proposed by our approach for this feature is *specify food complaint* and its description is "*this use case allows a citizen to register a food complaint. The food complaint has the following information: food complaint data, description and observations*".

From Table 7.12, we observe that the recall values are 100% of all the features that are documented. The recall values are an indicator for the efficiency of our approach. The values of precision are between [50% − 100%], which is high and its average is equal to 77%. F-Measure values rely on precision and recall values. The values of F-Measure are high too, between [66% − 100%] for the documented features and its average is equal to 85%.

Reducing the search space for use-cases and feature implementations across software variants is certainly the reason of this performance. In most cases, the contents of hybrid blocks are in the range of [3−6] use-cases and feature implementations. Another reason for this good result is that a *common vocabulary* is used in the use-case descriptions and feature implementations, thus lexical similarity was a suitable tool.

In Table 7.12, there are two feature implementations without documentations. These feature implementations are documented based on the OBE names. These feature implementations are: *complaint status* and *update medical speciality*. The reason behind this limitation is that these feature implementations don't have any lexical similarity with the corresponding use-cases in the given hybrid block. In this context, we document the mined feature implementations based on the *OBE names*. The proposed names for these features by using the *strongest weighted tokens* are "complaint status" and "update medical speciality".

The column (*k-Topics*) in Table 7.12 represents the number of term-topics. In our work, the number of k-Topics is equal to the number of feature implementations for each hybrid block. All feature names produced by our approach, in the column (*Feature Name*) of Table 7.12, represent the names of the use-cases.

The RCF corresponding to our approach contains two formal contexts and one relational context. The first formal context represents use-case diagrams. The second formal context represents feature implementations. The relational context of Health complaint-SPL is illustrated in Table 7.13. For Health complaint-SPL case study, two lattices of the CLF are represented in Figure 7.9.

Table 7.13 : The relational context for feature documentation of Health complaint software variants.

Relational context: appears-with	Special complaint impl.	Food complaint impl.	Animal complaint impl.	Update health unit impl.	Exception handling impl.	Persistence impl.	Complaint update impl.	Register new employee impl.	Update employee impl.	Query information impl.	Receive alerts via feeds impl.	Complaint status impl.	Update medical speciality impl.
Specify animal complaint	×	×	×	×	×	×							
Specify food complaint	×	×	×	×	×	×							
Specify special complaint	×	×	×	×	×	×							
Specify complaint	×	×	×	×	×	×							
Handle exception	×	×	×	×	×	×							
Persist data	×	×	×	×	×	×							
Update health unit	×	×	×	×	×	×							
Update employee							×	×	×	×			
Register new employee							×	×	×	×			
Update complaint							×	×	×	×			
Query information							×	×	×	×			
Receive alerts via feeds											×		

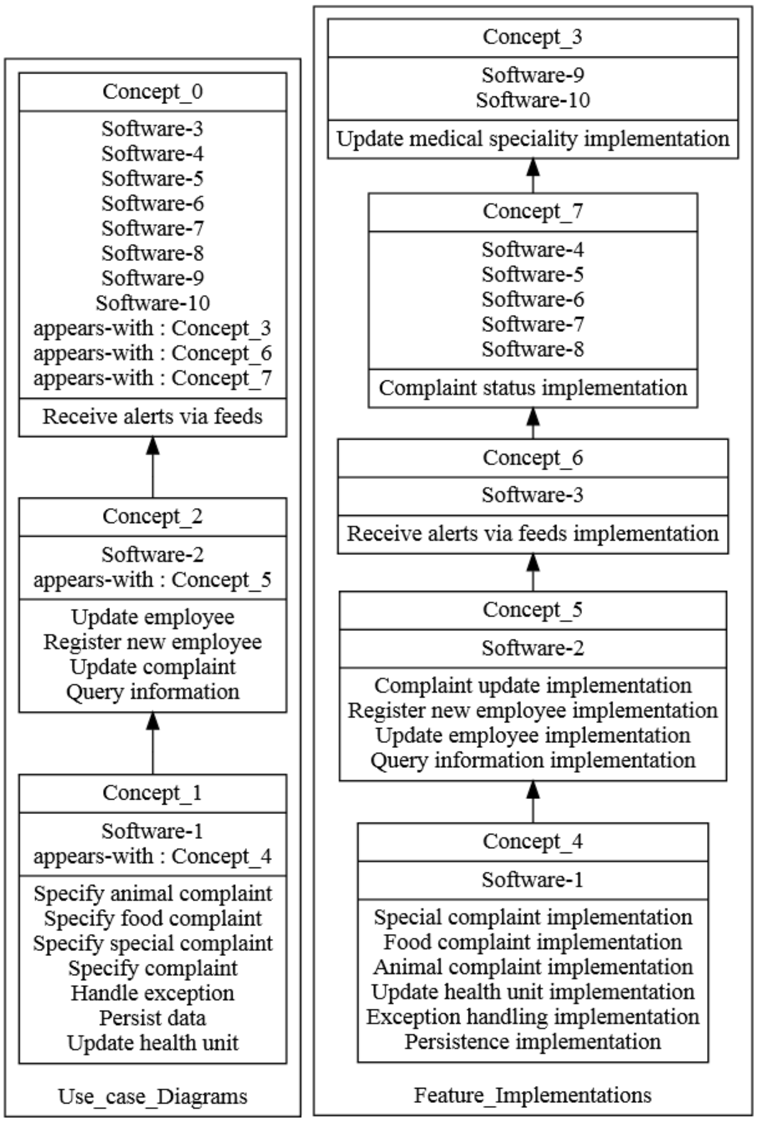


Figure 7.9 : The CLF of the RCF for features documentation of health complaint software variants.

7.3.2.3 FM Reverse Engineering

As we mentioned previously, the mined and documented features are stored in the product-by-feature matrix. Table 7.14 shows the product-by-feature matrix for health complaint software variants. The technique used to identify the FM relies on FCA. First, a formal context, where *objects* are product variants and *attributes* are features (*cf.* Table 7.14), is defined. The corresponding AOC-poset is then calculated (*cf.* Figure 7.10).

The technique used to identify the common and optional features in addition to the dependency between features relies on FCA. First, a formal context is defined. The corresponding AOC-poset is then calculated. The intent of each concept represents features common to two or more products. As concepts of AOC-posets are ordered, the intent of the most general (*i.e.* top) concept gathers features that are common to all product variants. They constitute the common features. The intents of all remaining concepts are optional features. They gather sets of optional features common to a subset of products or unique to a single product. The extent of each of these concepts is the set of products having these features in common (Figure 7.10).

Table 7.14 : The Product-by-feature matrix for Health complaint software variants.

	Public Health Complaint system SPL	Specify special complaint	Specify food complaint	Specify animal complaint	Update health unit	Handle exception	Persist data	Update complaint	Register new employee	Update employee	Query information	Receive alerts via feeds	Complaint status	Update medical speciality
Software-1	✖	✗	✗	✗	✗	✗	✗							
Software-2	✖	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓			
Software-3	✖	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓		
Software-4	✖	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	
Software-5	✖	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	
Software-6	✖	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	
Software-7	✖	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	
Software-8	✖	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	
Software-9	✖	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
Software-10	✖	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓

Product-by-feature matrix
(✖ the root feature; ✗ mandatory feature; ✓ optional feature)

Based on the AOC-poset in Figure 7.10, we identify the FM that represents the mined and documented features from health complaint. Figure 7.11 shows the mined FM for health complaint software variants. The resulting FM describes exactly the given product configurations. The needed time to identify the FM is equal to 464 ms.

The mined feature model includes three groups of features from an abstract type (added by our approach). The mandatory features consist of 6 concrete features. In the mined FM, we get a new mandatory feature which is the "update health unit" feature. This feature was not included in the original FM. Moreover, the extracted FM consists of 7 optional features. Two of these optional features were not included in the original FM. These features are "update medical speciality" and "complaint status" features.

The mined FM presents six cross-tree constraints of requires type. These constraints are obtained by parsing the AOC-poset. These constraints were not included in the original FM. We should mention that the resulting FM exactly describes the given product configurations.

Results show that precision appears to be not very high for Health complaint-SPL (*i.e.* 57%). This means that many of the identified product configurations of the mined FM are extra configurations (not in initial set that is defined by the original FM). Considering the recall metric, it is value is 100% for Health complaint-SPL. This means that product configurations defined by the initial FM are included in the product configurations derived from the mined FM. Considering the F-Measure metric, our approach has value 72%.

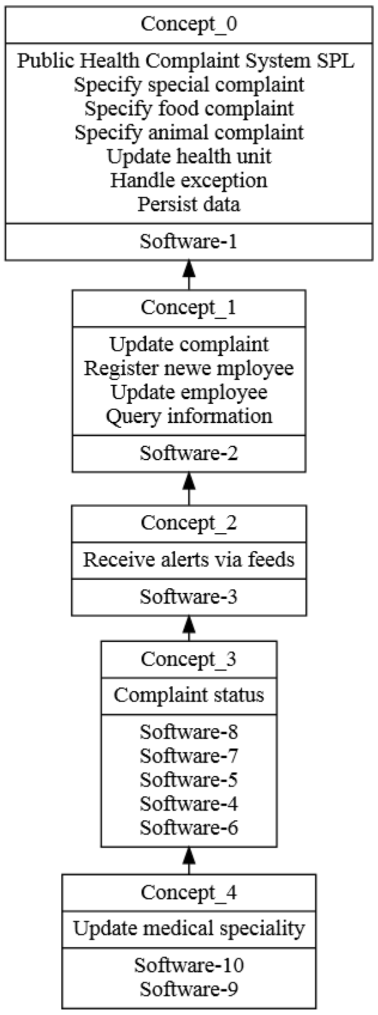


Figure 7.10 : The AOC-poset for the formal context of Table 7.14.

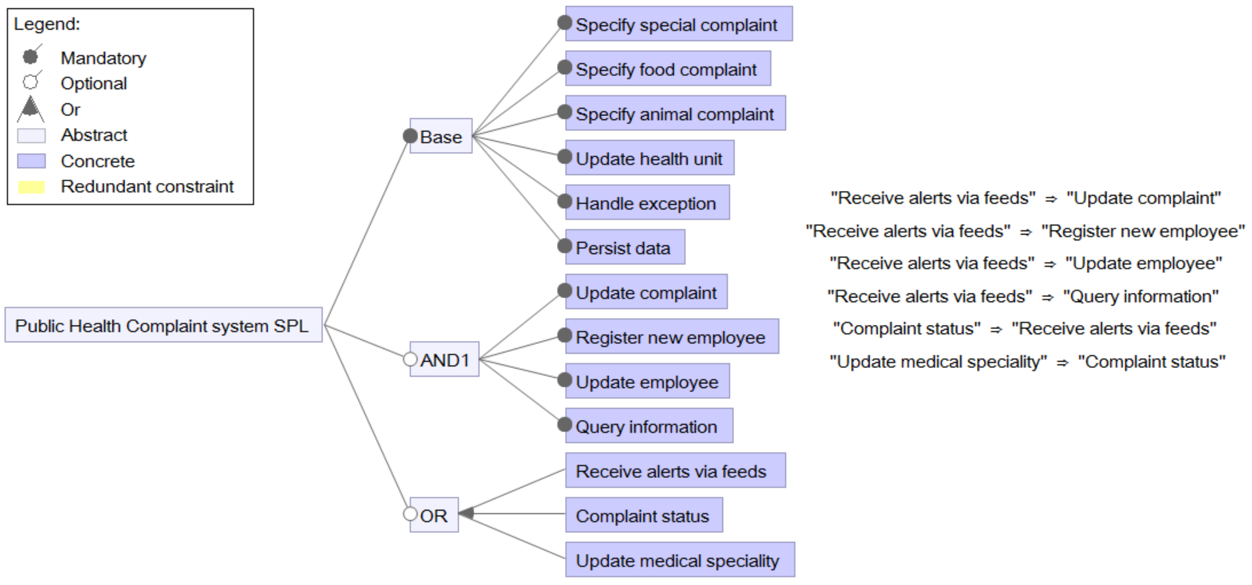

Legend:
Mandatory
Optional
Or
Abstract
Concrete
Redundant constraint
Public Health Complaint system SPL
Base
AND1
OR
Specify special complaint
Specify food complaint
Specify animal complaint
Update health unit
Handle exception
Persist data
Update complaint
Register new employee
Update employee
Query information
Receive alerts via feeds
Complaint status
Update medical speciality
"Receive alerts via feeds" ⇒ "Update complaint"
"Receive alerts via feeds" ⇒ "Register new employee"
"Receive alerts via feeds" ⇒ "Update employee"
"Receive alerts via feeds" ⇒ "Query information"
"Complaint status" ⇒ "Receive alerts via feeds"
"Update medical speciality" ⇒ "Complaint status"

Figure 7.11 : The extracted FM from Table 7.14.

7.4 Mobile Media Case Study

7.4.1 Mobile Media Description

Mobile Media[8] is a Java-based open source application that manipulates photo, music, and video on mobile devices, such as mobile phones [Tizzei *et al.*, 2011]. The system uses various technologies based on the Java ME platform. Table 7.15 summarizes the evolution scenarios in Mobile Media[9]. The first column describes the release in which the changes were applied. The third column describes the type of change and the type of feature (*e.g.* mandatory, optional) included or changed in that release. The scenarios comprise different types of changes involving mandatory and optional features, as well as non-functional concerns. For instance, release 1 implements the original system with just the functionality of viewing photos and organizing them by albums. Release 2 and 3 add error handling and the implementation of some optional features (sort photos by frequency or edit labels) respectively.

Table 7.15 : Summary of evolution scenarios in Mobile Media.

Release	Type of Change
R1	Mobile Photo core – mandatory features
R2	Inclusion of non-functional concern which is also an optional feature
R3	Inclusion of two optional features
R4	Inclusion of optional feature

We select 4 products of Mobile Media as published in [Tizzei *et al.*, 2011]. Naturally, the selected products cover all features of Mobile Media software variants. These variants consist of 8 features. Mobile Media variants are presented in Table 7.16 characterized by metrics LoC, NoP, NoC and number of OBEs.

Table 7.16 : Mobile Media software product variants.

Product #	Mobile Media Product Description	LoC	NoP	NoC	Number of OBEs
P1	Mobile photo - Base	936	7	15	822
P2	Exception handling included	1,213	8	24	925
P3	Sorting photos/edit photo label included	1,422	8	26	1,040
P4	New feature added to manage favourites	1,484	8	25	1,066

Figure 7.12 presents the FM of Mobile Media software variants. It shows the features of Mobile Media software variants, such as *add photo*, *delete photo* and *sorting*. To illustrate an evolution scenario, the *favourites* optional feature is shown in white circle above the box, because it was added in release 4. This FM is manually designed by the authors of Mobile Media software variants.

Figure 7.13 shows part of use-case diagram of release 3 from Mobile Media. In this figure, we show the new use-cases and the core use-cases that have relation with them. The new added artefacts (use-cases) are shown in blue color. We have just shown in the use-case diagram the changes added with respect to the original description of the system (in release 1). Listing 7.3 shows the description of the *view sorted photos* use-case in Figure 7.13.

[8] Mobile Media source code : `http://homepages.dcc.ufmg.br/~figueiredo/spl/icse08/`
[9] `http://www.ic.unicamp.br/~tizzei/mobilemedia/`

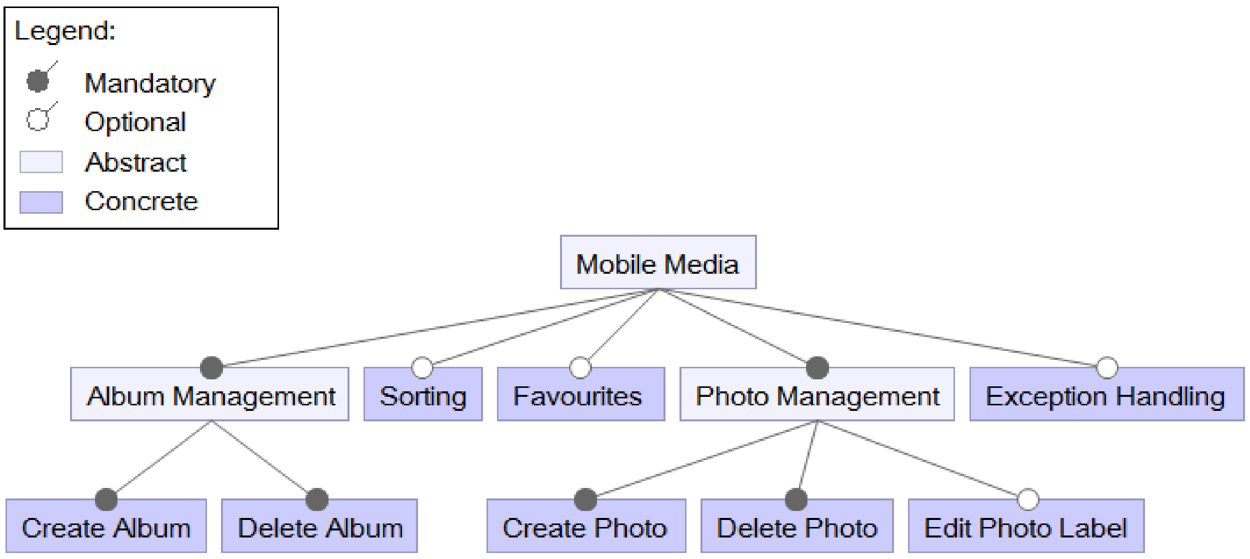


Figure 7.12 : Mobile Media feature model [Tizzei *et al.*, 2011].

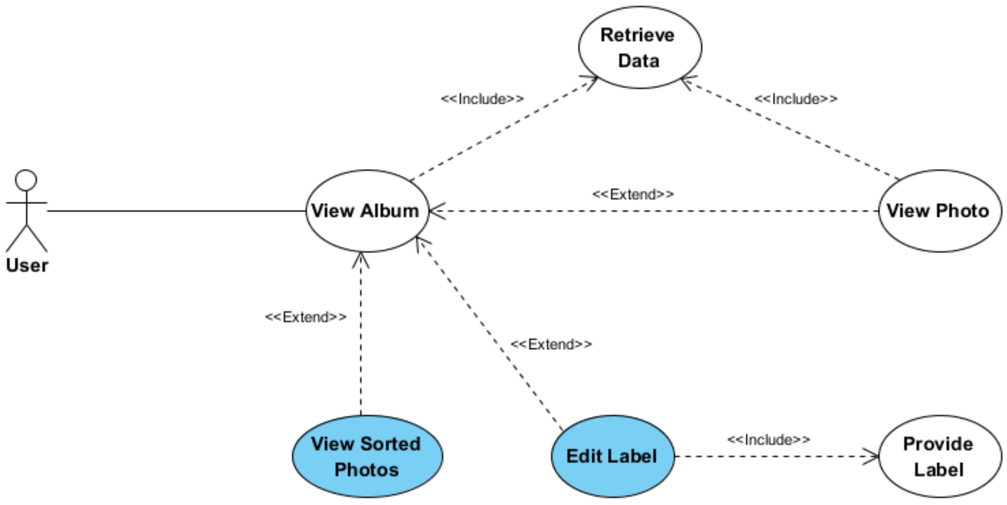


Figure 7.13 : Use case diagram for release 3 of Mobile Media [Conejero *et al.*, 2012].

```
Use-case: View Sorted Photos
Actor: Mobile Phone (system) and User.
Description:The device sorts the photos based on the number of times photo has been viewed.
Precondition:
1. Application must be launched.
2. A few photos must be available.
3. User must have selected the option to "select an album".
Post-condition: Photos are displayed sorted by frequency.
Trigger: User has selected the option to "sort by view" from the album menu.
Basic flow:
1. The user selects the option to "Sort by view".
2. The device populates the list of photos sorted by the highest viewing frequency.
Alternative flows:
2. The device is unable to access to the frequency of the photos.
Extends: View Album use-case.
```

LISTING 7.3 : The description of the *view sorted photos* use-case in Figure 7.13.

7.4.2 Mobile Media Validation

7.4.2.1 Feature Location

Table 7.17 summarizes the obtained results for Mobile Media case study. For readability's sake, we manually associated feature names to atomic blocks (*i.e.* feature implementations). For Mobile Media software variants, we rely only on the *lexical similarity* to identify their features. The reason behind this choice is that variability expresses at different levels (*e.g.* package, class, attribute, method and method body) not only at class level.

Table 7.17 : Features mined from Mobile Media software product variants.

#	Feature Name	Feature Type		k-Topics	Evaluation Metrics		
		Mandatory	Optional		Precision	Recall	F-Measure
1	Create album	✗		6	81%	58%	67%
2	Delete album	✗		6	80%	62%	69%
3	Create photo	✗		6	81%	52%	63%
4	Delete photo	✗		6	78%	63%	69%
5	View album	✗		6	87%	68%	76%
6	Retrieve data	✗		6	71%	57%	63%
7	Exception handling		✗	1	100%	70%	82%
8	Edit photo label		✗	2	100%	77%	87%
9	Sorting		✗	2	100%	78%	87%
10	Favourites		✗	1	100%	80%	88%

Results show that we get all features of FM in Figure 7.12 in addition to *view album* and *retrieve data* features. Results show that *precision* appears to be high for all optional features. This means that all mined OBEs grouped as features are relevant. This result is due to search space reduction. In all cases, each BV corresponds to one and only one feature implementation.

For mandatory features, *precision* is also quite high thanks to our clustering technique that identifies atomic block based on FCA and LSI. However, *precision* is smaller than the one obtained for optional features. This deterioration can be explained by the fact that we do not perform search space reduction for the common block (CB). The common block contains implementations for all mandatory features. We apply directly the LSI on the common block. In all cases, our approach shows *excellent* results to detect software variability. The mandatory features appear in all software variants; so we believe that identification of common OBEs like common block is more useful than split the common block.

Considering the *recall* metric, its average value is 66% for Mobile Media. This means that most OBEs that compose features are mined. We have manually identified OBEs which should have been mined and were not. We found that these non-mined OBEs used different vocabularies from mined OBEs'. This is a known limitation of LSI which is based on lexical similarity. Considering the *F-Measure* metric, our approach has values that range from 63% to 88%. This means that most OBEs that compose features are mined and shows the efficiency of our approach.

In our work we cannot use a fixed number of *k-Topics* for LSI because we have blocks of variation (*i.e.* partitions) with different sizes (*cf.* Table 7.17). For the common block, we use a fixed number for the k-Topics of LSI method. Listing 7.4 shows part of the *favourites* feature implementation. Each feature implementation consists of a set of OBEs (*i.e.* set of attributes, methods, method invocations and attribute accesses).

```
Attribute Access (length_showImageList(recordName_sort_favorite))
Attribute Access (viewFavoritesCommand_initMenu())
Attribute Access (favorite_setFavorite(favorite))
Attribute (viewFavoritesCommand_PhotoListScreen)
Method Invocation (addCommand[viewFavoritesCommand]_initMenu())
Attribute (viewFavoritesCommand_FavouritesList)
Method (setFavorite(favorite)_FavouritesList)
Method (setFavorite(favorite)_ImageData)
Method Invocation (equals["ViewFavorites"]_handleCommand(c_d))
Method Invocation (equals["SetFavorite"]_handleCommand(c_d))
Method Invocation (setFavorite[favorite]_toggleFavorite())
                                          .
                                          .
```

LISTING 7.4 : Part of the favourites feature implementation.

7.4.2.2 Feature Documentation

In order to document the mined feature implementations from Mobile Media software variants, we rely on the use-case diagrams of these variants. Table 7.18 summarizes the obtained results (*i.e.* feature names). For instance, the example below shows the name and description of *edit label* feature from Mobile Media software variants.

Example 7.3. *Edit Label : "The user can edit the existing label of photo and album".*

Table 7.18 : Features documented from Mobile Media software product variants.

#	Feature Name	Hybrid block #	k-Topics	Evaluation Metrics		
				Recall	Precision	F-Measure
1	Delete album	HB−1	6	100%	50%	66%
2	Delete photo	HB−1	6	100%	50%	66%
3	Add album	HB−1	6	100%	50%	66%
4	Add photo	HB−1	6	100%	50%	66%
5	View album	HB−1	6	100%	100%	100%
6	Retrieve data	HB−1	6	100%	100%	100%
7	Exception handling	HB−2	1	100%	100%	100%
8	View sorted photos	HB−3	2	100%	50%	66%
9	Edit label	HB−3	2	100%	50%	66%
10	Set/View favourites	HB−4	1	100%	100%	100%

For Mobile Media case study, we observe that the *recall* values are 100% of all the features that are documented. The recall values are an indicator for the efficiency of our approach. The values of *precision* are between [50% − 100%], which is high. F-Measure values rely on precision and recall values. The values of F-Measure are high too, between [66% − 100%] for the documented features.

Reducing the search space for use-cases and feature implementations across software variants is certainly the reason of this performance (*cf.* Figure 7.14). In most cases, the contents of hybrid blocks are in the range of [1 − 6] use-cases and feature implementations. Another reason for this good result is that a common vocabulary is used in the use-case descriptions and feature implementations, thus lexical similarity was a suitable tool.

The column (*k-Topics*) in Table 7.18 represents *the number of term-topics.* All feature names produced by our approach, in the column (*Feature Name*) of Table 7.18, represent the names of the use cases. For example, in the FM of Mobile Media [Tizzei *et al.*, 2011] there is a feature called *sorting*. The name proposed by our approach for this feature is *view sorted photos* and its description is "*the device sorts the photos based on the number of times photo has been viewed*".

Table 7.19 : The relational context for feature documentation.

Relational context: appears-with	Create album impl.	Delete album impl.	Create photo impl.	Delete photo impl.	View album impl.	Retrieve data impl.	Exception handling impl.	Edit photo label impl.	Sorting impl.	Favourites impl.
View Album	×	×	×	×	×	×				
Add Photo	×	×	×	×	×	×				
Delete Photo	×	×	×	×	×	×				
View Photo	×	×	×	×	×	×				
Add Album	×	×	×	×	×	×				
Delete Album	×	×	×	×	×	×				
Provide Label	×	×	×	×	×	×				
Store Data	×	×	×	×	×	×				
Remove Data	×	×	×	×	×	×				
Retrieve Data	×	×	×	×	×	×				
Exception handling							×			
View Sorted Photos								×	×	
Edit Label								×	×	
View Favourites										×
Set Favourites										×

The RCF corresponding to our approach contains two formal contexts and one relational context. The first formal context represents the use-case diagrams. The second formal context represents feature implementations. The relational context of Mobile Media illustrated in Table 7.19. For Mobile Media case study, two lattices of the CLF are represented in Figure 7.14.

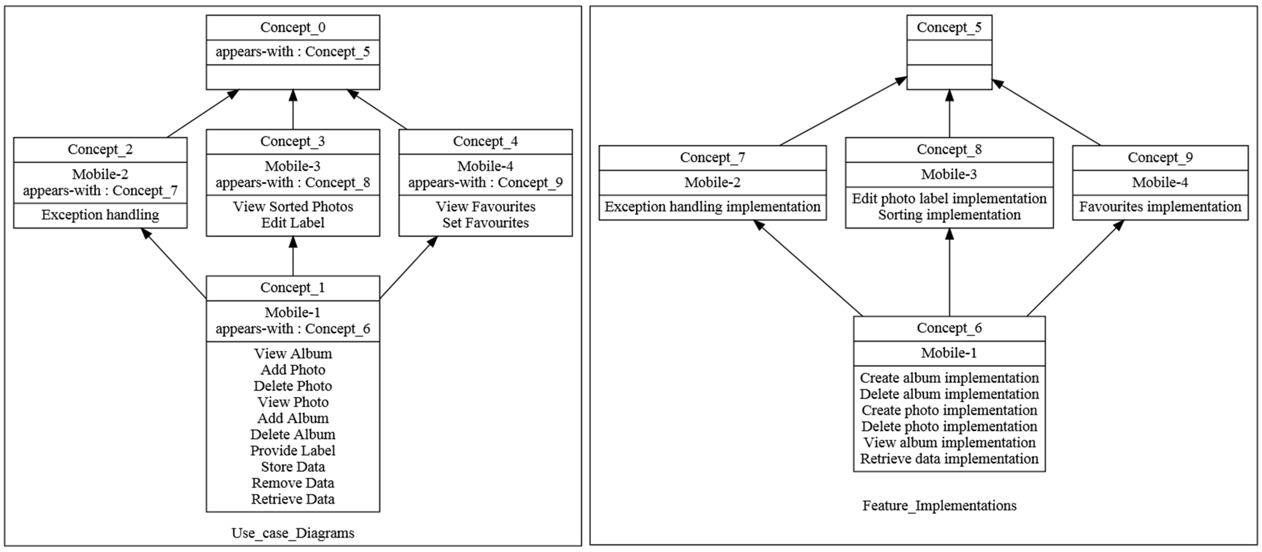


Figure 7.14 : The concept lattice family of the relational context family for features documentation.

7.4.2.3 FM Reverse Engineering

The mined and documented features for Mobile Media are stored in the product-by-feature matrix. Table 7.20 shows the product-by-feature matrix for Mobile Media software variants. We rely on FCA to identify the FM of these software variants. First, a formal context, where *objects* are product variants and *attributes* are features (*cf.* Table 7.20), is defined. The corresponding AOC-poset is then calculated (*cf.* Figure 7.15).

Table 7.20 : The Product-by-feature matrix for Mobile Media software product variants.

	Mobile Media	Delete album	Delete photo	Add album	Add photo	View album	Retrieve data	Exception handling	View sorted photos	Edit label	Set/View favourites
Release−1	✖	✗	✗	✗	✗	✗	✗				
Release−2	✖	✗	✗	✗	✗	✗	✗	✓			
Release−3	✖	✗	✗	✗	✗	✗	✗		✓	✓	
Release−4	✖	✗	✗	✗	✗	✗	✗				✓

Product-by-feature matrix
(✖ the root feature; ✗ mandatory feature; ✓ optional feature)

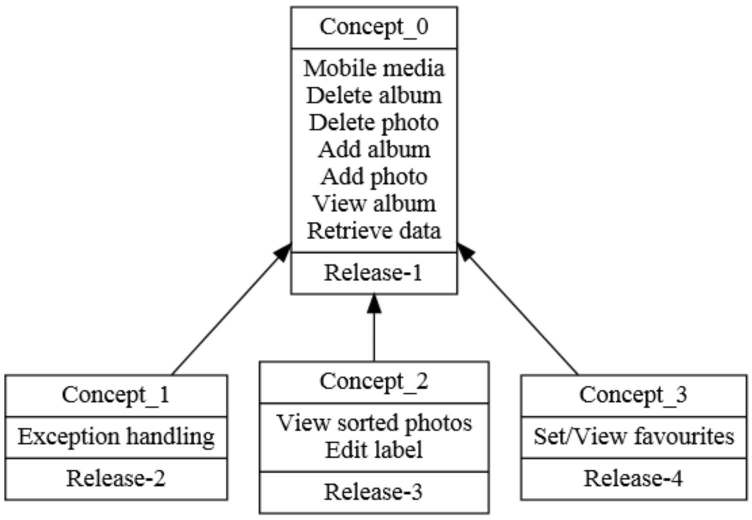


Figure 7.15 : The AOC-poset for the formal context of Table 7.20.

Based on the AOC-poset in Figure 7.15, we identify the FM which is representing the mined and documented features from Mobile Media case study. Figure 7.16 shows the mined FM for Mobile Media. The resulting FM describes exactly the given product configurations. The needed time to identify the FM is equal to 441 ms. We should mention that the extracted FM in Figure 7.16 was built using the REVPLINE approach with our *notations* (see legend of the Figure 7.16).

The mined FM is correct based on the given formal context. The original FM of Mobile Media consists of: two abstract features, eight concrete features (*i.e.* four optional features and four mandatory features), three levels of hierarchy and no cross-tree constraints. The REVPLINE

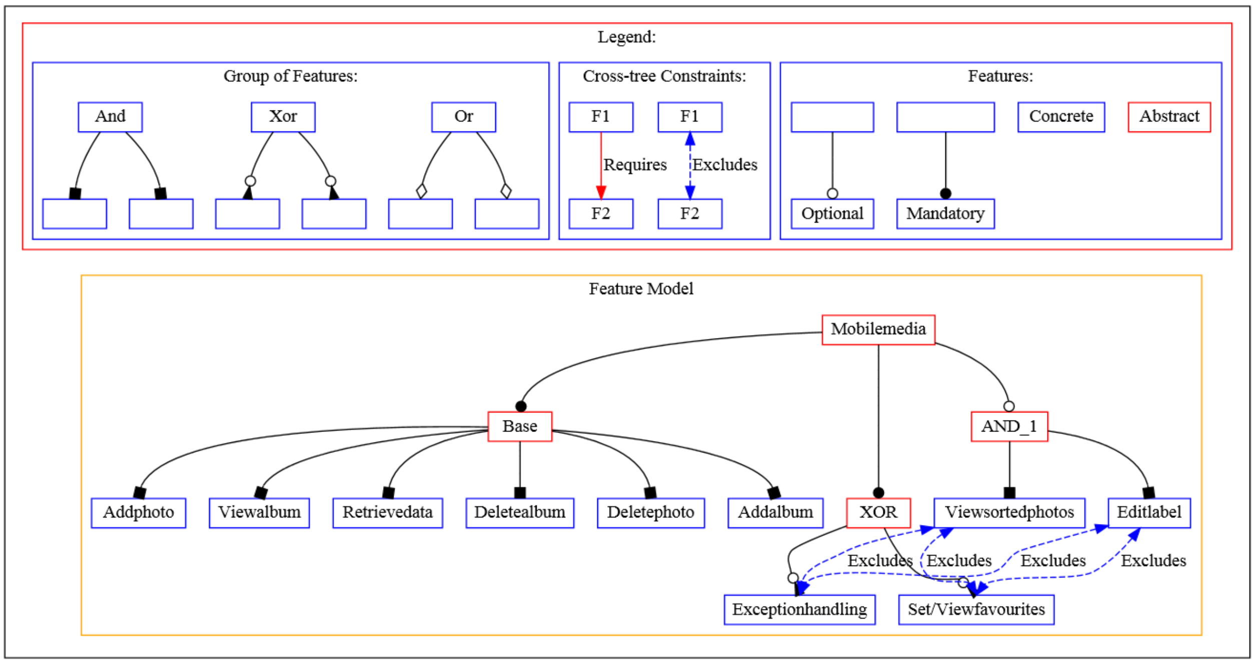


Figure 7.16 : The extracted FM from Table 7.20.

approach identifies ten feature implementations from Mobile Media source code. The extracted FM consists of ten concrete features (*i.e.* four optional features and six mandatory features) and four cross-tree constraints.

The feature names in the mined FM are very close to the feature names in the original FM. For instance, in the original FM there is a feature called "*sorting*", while the name becomes "*view sorted photos*" in the mined FM. Our approach extracts concrete features from a collection of software product variants. The mined FM identifies the dependency between the mined features.

Listing 7.5 shows the mined FM as SXFM format[10]. SXFM is for simple XML feature model format used in SPLOT homepage[11]. The REVPLINE generates FM in several formats such as FeatureIDE, Dot file and SXFM format. The SXFM format can be used online through SPLOT's feature model editor. To get all information from SXFM file, Java parser is available for reading the SXFM file at SPLOT homepage.

We can note that the extracted FM by our approach presents a unique and consistent FM. Moreover, the mined FM is introducing new constraints that do not exist in the original FM (the manually designed FM by the authors of Mobile Media case study). Figure 7.17 shows the extracted FM from Mobile Media via FeatureIDE plugin [Thüm *et al.*, 2014].

Results show that precision appears to be not very high for ArgoUML-SPL (*i.e.* 68%). This means that many of the identified product configurations of the mined FM are extra configurations (not in initial set that is defined by the original FM). Considering the recall metric, it is value is 100% for ArgoUML-SPL. This means that product configurations defined by the initial

[10]Available at : http://gsd.uwaterloo.ca:8088/SPLOT/models/temp_models/model_20140321_460214112.xml
[11]http://www.splot-research.org/

```
<feature_model name="Mobile Media">
<feature_tree>
:r Mobile Media(_r)
   :m Base(_r_1)
   :m Delete album(_r_1_3)
   :m Delete photo(_r_1_4)
   :m Add album(_r_1_5)
   :m Add photo(_r_1_6)
   :m View album(_r_1_7)
   :m Retrieve data(_r_1_8)
   :o AND 1(_r_10)
      :m View sorted photos(_r_10_11)
      :m Edit label(_r_10_12)
   :g (_r_16) [1,1]
      : Exception handling(_r_16_17)
      : Set-View favourites(_r_16_21)
</feature_tree>
<constraints>
constraint_1:~_r_10_11 or ~_r_16_17
constraint_4:~_r_10_12 or ~_r_16_21
constraint_2:~_r_10_12 or ~_r_16_17
constraint_3:~_r_10_11 or ~_r_16_21
</constraints>
</feature_model>
```

LISTING 7.5 : The mined FM as SXFM format.

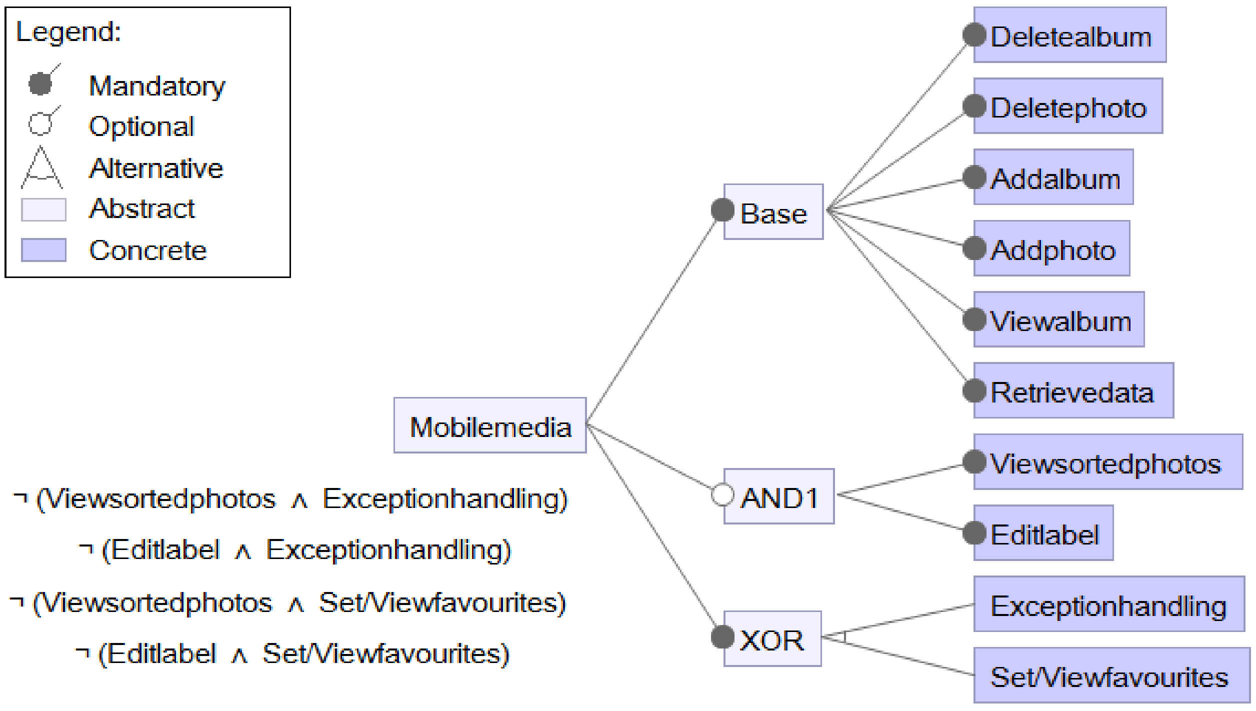


Figure 7.17 : The mined FM from Mobile Media via FeatureIDE plugin.

FM are included in the product configurations derived from the mined FM. Considering the F-Measure metric, our approach has value 80%.

7.5 Reverse Engineering FMs from Samples of Program Configurations

The goal of this section is to show other results obtained by our approach[12] from a set of program configurations (samples). We ran experiments on 9 case studies. The number of products and features vary from a case study to another. In addition, these case studies are well known in SPL domain and well documented; their feature models are available for comparison to our results so as to validate our proposal.

We ran experiments on 9 product configurations: video on demand-SPL [Haslinger *et al.*, 2011] [Acher *et al.*, 2013a], wiki engines [Acher *et al.*, 2012], mobile phone [Davril *et al.*, 2013], DC motor [Ryssel *et al.*, 2011], berkeley-DB [Xue, 2013], graph Product Line [Lopez-Herrejon and Batory, 2001], Wikipedia [Bécan *et al.*, 2013], cell phone-SPL [Haslinger, 2012] and sample set of clock configurations [She, 2008]. In the last two cases, our goal was only to show the scalability of our approach (without evaluation metrics). We generated large matrices and we used them as input of our approach.

Table 7.21 summarizes the results obtained. We present our results according to some criteria such as number of products and features, base, -and, xor and or constraints. In addition to the cross-tree constraints (requires and excludes). Finally, we present the execution time for each case study.

Table 7.21 : The results of the product configurations that are identified by the mined FMs.

				Group of Features				CTCs			Evaluation Metrics		
#	case study	Number of Products	Number of Features	Base	Atomic Set of Features	Inclusive-or	Exclusive-or	Requires	Excludes	Algorithm execution times \ (in ms)	Precision	Recall	F-Measure
1	Video on demand	16	12	✗	✗	✗		✗		572	66%	100%	80%
2	Wiki engines	8	21	✗	✗	✗	✗	✗	✗	555	54%	100%	70%
3	Graph product line	8	18	✗		✗	✗	✗	✗	551	62%	100%	76%
4	Berkeley DB	10	43	✗	✗	✗	✗	✗	✗	661	50%	100%	66%
5	Mobile phone	5	5	✗		✗		✗		406	70%	100%	82%
6	DC motor	10	15			✗		✗		444	83%	100%	90%
7	Wikipedia	10	14	✗	✗	✗		✗		552	72%	100%	84%
8	Cell phone-SPL	16	13	✗	✗	✗	✗	✗	✗	486	51%	100%	68%
9	Clock	4	6	✗		✗		✗		486	60%	100%	75%
10	1000 × 27 matrix	1000	27	✗	✗	✗	✗	✗	✗	46811	-	-	-
11	1500 × 137 matrix	1500	137	✗	✗	✗	✗	✗	✗	60350	-	-	-

For correctness, we performed a straightforward test based on the semantics of FM which is defined as the set of their accepted configurations. We computed the list of products of our

[12]Source code: `https://code.google.com/p/refmfpc/`

reversed-engineered feature models, again using the FAMA tool suite[13], and compared them with the list of products originally used as input.

Results show that precision appears to be not very high for all case studies. This means that many of the identified product configurations of the mined FM are extra configurations (not in initial set that is defined by the original FM). For the precision metric, our approach has values that range from 50% to 83%. Considering the recall metric, it is value is 100% for all case studies. This means that product configurations defined by the initial FM are included in the product configurations derived from the mined FM. Considering the F-Measure metric, our approach has values that range from 66% to 90%. This means that our approach is efficient in many cases.

Theory and experiments show that if the generated AOC-poset has only one bottom concept (containing one product) there is no exclusive-or relation or exclude constraints from the given product configurations. Time needed to extract the FM (*cf.* Table 7.21) in *ms* includes all processes (starting from converting the product configurations to formal context and generate the AOC-poset, in addition to analyse the AOC-poset and generate the FM). These execution time was always reasonable on real SPL.

7.6 Conclusion

In this chapter, we presented a number of experiments to validate our approach. The experiments have proven the effectivity of our approach for:

❶ Identifying the implementation of a feature as a set of source code elements (*i.e.* OBEs), which enable a quick understanding of the software variant features and facilitate the maintenance tasks.

❷ Documenting the mined feature implementations by assigning a name and a description for each feature implementation. The feature documentation process enables a quick understanding of the feature functionality (feature comprehension) and explains the role of this feature in software product variants.

❸ Reverse engineering of the feature model based on the mined and documented features. Identifying the constraints (dependencies) between features of the feature model is an important task. The identified feature model is assisting the expert by better understanding of the software members, software features and constraints between features.

❹ The REVPLINE approach allows us to achieve the most important tasks in order to re-engineering software product variants into a SPL for systematic reuse: ① identifying the feature implementations, ② documenting the identified feature implementations and ③ synthesizing the feature model (feature dependencies).

❺ In our approach we consider only Java software systems. This is representing a *threat* to prototype validity that limits our implementation ability to deal only with software variants that are developed based on Java language. There are other threats to validity that are mentioned in the previous chapters (*cf.* Chapter 4, Chapter 5 and Chapter 6).

In general, we showed that our approach has assisted the construction of SPLs from existing software variants, by facilitating the mining and documenting the features from the source

[13] http://www.isa.us.es/fama/

code of these variants. Finally, the mined and documented features are expressed as the feature model which showed the constraints between features.

For all case studies, the feature mining results validated the relevance and the performance of our proposal as most of the features were correctly identified. The feature documentation results of this evaluation showed that most of the features have been documented correctly. The FM reverse engineering results showed that most of the features and their associated constraints are correctly identified. The evaluation metrics values (precision, recall and F-measure) showed that the obtained results are not equivalent in the three case studies.

Part IV

Conclusion and Perspectives

CHAPTER 8

Conclusion and Future Directions

The value of a man resides in what he gives and not in what he is capable of receiving.

Albert EINSTEIN

Preamble

In this chapter, we summarize our main contributions and discuss future directions of research. We summarize the results and conclusions of the dissertation. We also discuss opportunities for extending our work. In Section 8.1, we present the main contributions of REVPLINE approach. Section 8.2 presents the future directions of our approach.

8.1 Summary of Contributions

The main objective of this dissertation is to re-engineering software variants into a SPL. This problem is crucial and one of the most studied issues in the field of SPLE. We present here a summary of our contributions as follows.

❶ **State of the Art.** In Chapter 3, we reviewed the techniques that have been proposed for building a SPL from existing software variants. First, we introduced and discussed the main approaches of feature location. Then, we discussed and gave an overview of the main approaches related to source code comprehension. Finally, we discussed the existing approaches for reverse engineering FMs. In each category we present a comparison of the selected approaches based on a set of criteria that we consider relevant to our contributions. At the end of each contribution chapter, we add a summary of our approach with the same criteria.

❷ **Mining features from the source code of a set of software variants.** We proposed in Chapter 4 the first contribution of the REVPLINE, which is a new approach for feature location in a collection of software product variants. REVPLINE extracts features by using lexical or structural similarity. In this approach, we aim at providing an approach based on several techniques such as FCA, LSI and source code dependency in order to contribute in providing a solution for feature location in a collection of software product variants. In our approach we exploit commonality and variability across software variants at source code level to apply the LSI method in an efficient way. In our feature location approach each feature implementation corresponds to a set of OBEs.

❸ **Documenting the mined feature implementations.** We proposed in Chapter 5 the second contribution of the REVPLINE: a new approach for feature documentation. Our approach aims to document the mined feature implementations by giving names and descriptions, based on the feature implementations and use-case diagrams of software variants. The novelty of our approach is that it exploits commonality and variability across software variants, at feature implementations and use-cases levels, to run Information Retrieval methods in an efficient way. The features documentation process is based on the source code and on the use-case diagrams of software variants. Feature implementations can be documented also using the names of the OBEs corresponding to atomic blocks when use-cases are missing. We aim at providing an approach based on several techniques (*i.e.* FCA, LSI and RCA).

❹ **Reverse engineering FM from the mined and documented features.** We proposed in Chapter 6 the third contribution of the REVPLINE approach. We presented a new approach to organize the mined documented features into a feature model. Features are organized in a tree which highlights mandatory features, optional features and feature groups (and, or, xor groups). The feature model is completed with requirement and mutual exclusion constraints. We rely on Formal Concept Analysis and software configurations to mine a unique and consistent feature model. The resulting FM covers exactly the given software configurations.

❺ We presented a number of experiments to validate our approach in Chapter 7. The experiments have proven the effectivity of REVPLINE for: mining a set of features from the source code of software variants as a set of atomic blocks where each block consists of a set of OBEs, associating a documentation (name and description) for each feature implementation to enable a quick understanding of feature functionality and assisting an expert in building real FMs, based on the mined and documented features.

❻ To fully validate our proposal, we implemented a prototype of REVPLINE approach. REVPLINE implementation is deployed as a standalone application. In Appendix A, we presented our implementation. Firstly, we gave the architecture of REVPLINE in general as a component diagram. Then, we presented each component and its role. Finally, we presented the statistical information regarding REVPLINE source code such as number of packages and classes.

In order to find our position between the selected work presented in state of the art Chapter 3, we reconsider the criteria presented in this chapter, and we integrate our work in it (Thesis REVPLINE), as we can see in Table 8.1. The formal context consists of *articles* as objects and the *criteria* as attributes. In this table, we show part of this formal context that relates to our approach.

Table 8.1 : The selected criteria used to categorize the selected work with REVPLINE approach.

	Target Software		Java Source Code	Feature Location	Code-To-Feature Traceability Link	Source Code Comprehension	Feature Documentation		Reverse Engineering FMs		Future Directions		
	Single Software	Software Family					Automatically	Manually	High Level Models	Low Level Models	Dynamic Analysis	Feature Evolution	Impact Analysis
Thesis REVPLINE		✘	✘	✘		✘	✘			✘			
......													

We generate the corresponding lattice, shown in Figure 8.1, which clarifies our position according to the other works. We can notice that for our fixed objectives, there is not any work that meets them all together (no concepts below Thesis REVPLINE). We notice also that we still miss supporting feature evaluation, dynamic analysis and impact analysis.

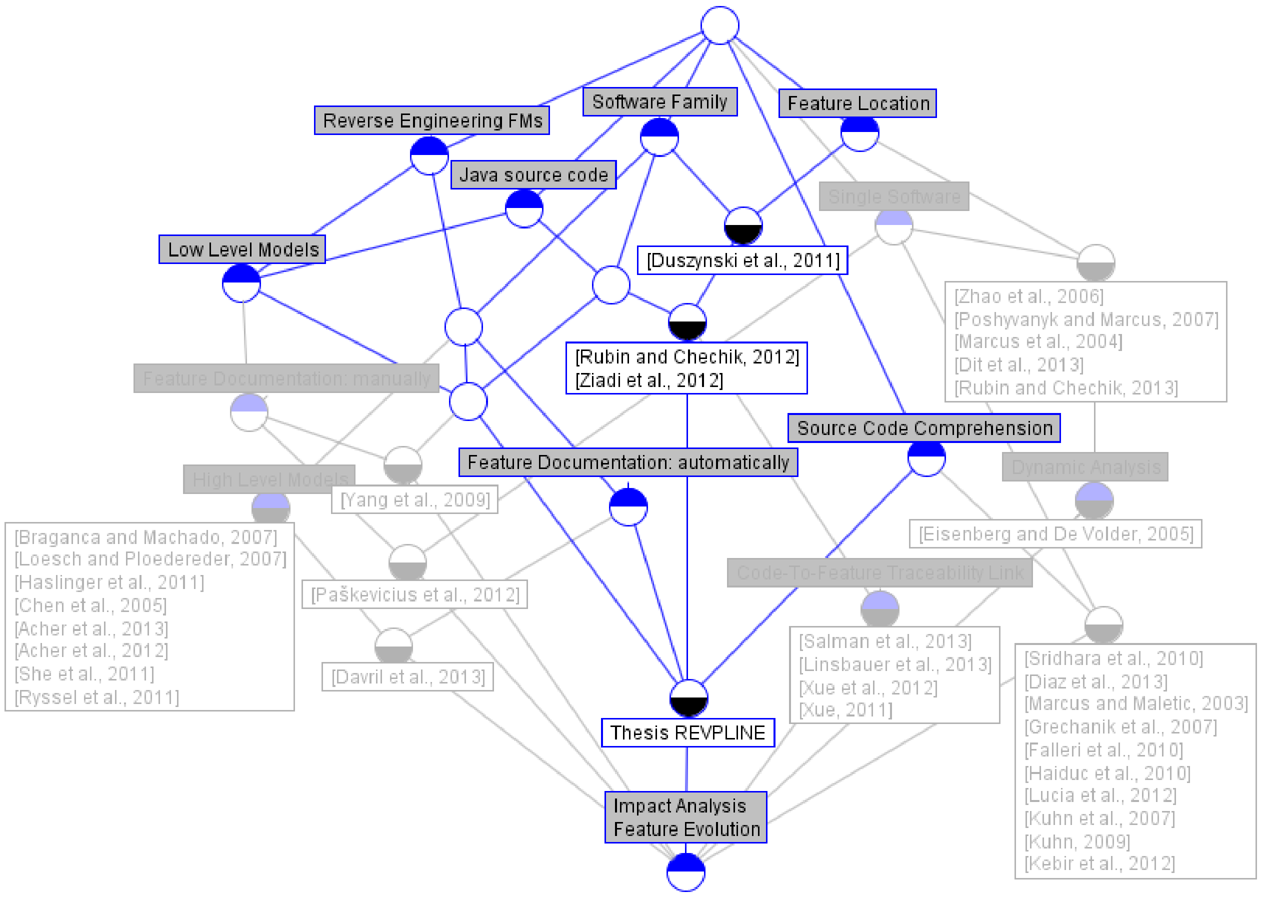


Figure 8.1 : Our position in the lattice of related work.

8.2 Future Directions

Here, we present a list of directions of research - regarding re-engineering of software variants into SPL - which we believe to be interesting to explore.

8.2.1 Extending the Scope

- To re-engineer an existing family of software variants into a SPL, several important prerequisites must be satisfied. First, commonality and variability among the product variants should be explicitly identified and must be systematically managed (*our current work*). Second, we should be able to derive a new software product from reusable components, so-called SPL core assets (*future direction*).From this identification of commonality and variability among the products, as they provide a basis for scoping an SPL, we can design first-cut SPL core assets. A wide range of variability techniques can be applied to design such SPL core assets from our knowledge of each feature and its implementation. The role of variability techniques such as *Java conditional* [Xue, 2013] is to make core assets reusable in multiple product variants. Variability should make such adaptive reuse easy. Such a process should allow the experts to drive new products based on the built core assets and mined FMs.

- In our work, we don't consider feature evolution. A feature always has the same implementation in all software product variants where it is present. As a perspective of our work, we plan to study the feature evolution in a family of software product variants. Actually, in our work we study the software variants which differ only in term of added or removed features. But when we consider the last 4 versions of mobile media we note that all mobile photo software variant features evolve with time. For the first 4 versions of mobile media we deal only with photo after that we deal with media such as photo, music, video in the recent versions. For example, feature "*edit photo label*" in release 3 evolves to be "*edit media label*" in release 8 of mobile media. The advantage of our current work is that we detect any level of variability except if the name of OBE changes from version to another version. There are needs to study this topic in more details.

- REVPLINE assumes that commonality and variability between software product variants can be determined statically, such as software product variants of the mobile media used in our evaluation. However, there exist systems that only behave differently depending on runtime parameters. For such software variants, we need to extend our approach with dynamic analysis techniques in order to obtain feature implementations of different software product variants at runtime.

8.2.2 Improving the Approach with Natural Language Processing Tools

- The current work extracts a FM which consists of two levels of hierarchy. As a perspective of this work, we plan to enhance the extracted FM by increasing the levels of hierarchy in addition to splitting the group of features into a set of groups based on the lexical or structural similarity. We also plan to automatically propose names for the parent features to reflect the roles of its features.

- In the current work, we mainly rely on the OBE names and use-cases to documenting the mined feature implementations. As a perspective of this work, we plan to use part of speech tagging to document the mined features directly from feature implementations. Our goal is to develop a cluster-naming process that involves selecting the most frequently occurring phrases from among all of the OBE names in the cluster (feature implementation). To identify the most frequently occurring phrases we will use the Stanford Part-of-Speech (POS) tagger[1] to tag each term in the OBE names with its POS. The OBE names are then pruned to retain only nouns, adjectives, verbs and other terms. Frequent item-sets are then discovered for each of the clusters. In our context, frequent item-sets are sets of terms that frequently co-occur together in the OBE names assigned to the same cluster. To select the name for a cluster, all of its frequent item-sets of maximum size, FIS_{max} are selected. Next, all OBE names in the cluster are examined. In each OBE name, the shortest sequence of terms which contains all the words in FIS_{max} is selected as a candidate name. Finally, the shortest candidate name is selected, as this reduces the verbosity of the feature name.

8.2.3 Improving techniques

- In the current work, we mainly rely on FCA and RCA as clustering techniques. As a perspective of this work, we plan to use search based algorithms as clustering technique instead of FCA and RCA. Furthermore, there is a limitation of using basic FCA as clustering technique. Basic FCA deals with binary formal context $(1, 0)$ and this affects on the quality of

[1] http://nlp.stanford.edu/software/tagger.shtml

the result. For example, the similarity value 0.99 is equal to 0.70 and 0.69 is equal to 0. There are FCA approaches (*cf.* [Kaytoue *et al.*, 2010]) that allows us to use numerical similarity values and avoid the problem Abe-books web-have basic with FCA. We plan to apply a meta-heuristic algorithm on both common block and blocks of variation to identify subgroups of elements that characterize the implementation of each possible feature based on the lexical and structural similarity between the OBEs of each block.

- For the current feature location approach, we have not identified junctions between features as junctions. We consider junction between features as feature in current work. As perspective of this work, we plan to distinguish features from the junctions. From the experience on ArgoUML we design such a principle: a junction corresponds to a set of OBEs that are used to implement more than one feature. In the source code, OBEs that implement a feature can use OBEs from a junction, but not the reverse.

APPENDIX A

IMPLEMENTATION

Make it work.
Make it work right.
Make it work right and fast.
Edsger DIJKSTRA, DonaldKNUTH, C.A.R. HOARE

Preamble

In this appendix, we describe our implementation of REVPLINE, which we developed to validate our proposal. Section A.1 gives an introduction of this appendix and shows the architecture of REVPLINE in general as a component diagram. Then, we explain all components of REVPLINE in Section A.2: we present input, output and role of each component. Section A.3 presents the external libraries that we used in REVPLINE. Section A.4 presents the eclipse plugins that we used in REVPLINE. In Section A.5, we present the statistical information regarding REVPLINE source code such as the number of packages and classes. Finally, in Section A.6, we conclude this appendix.

A.1 Introduction

In this appendix, we present the REVPLINE implementation[1]. We show a simplified structural view of the architecture of REVPLINE as a component diagram. Furthermore, we describe each component input/output in addition to its task. Then we present statistical information regarding REVPLINE source code such as number of packages, classes and so on.

Figure A.1 provides an overview of the REVPLINE architecture. In the next section, we describe each component and explain the role of this component in REVPLINE approach. We also present the used *plugins* and *libraries* and explain the role of each plugin/library in our work.

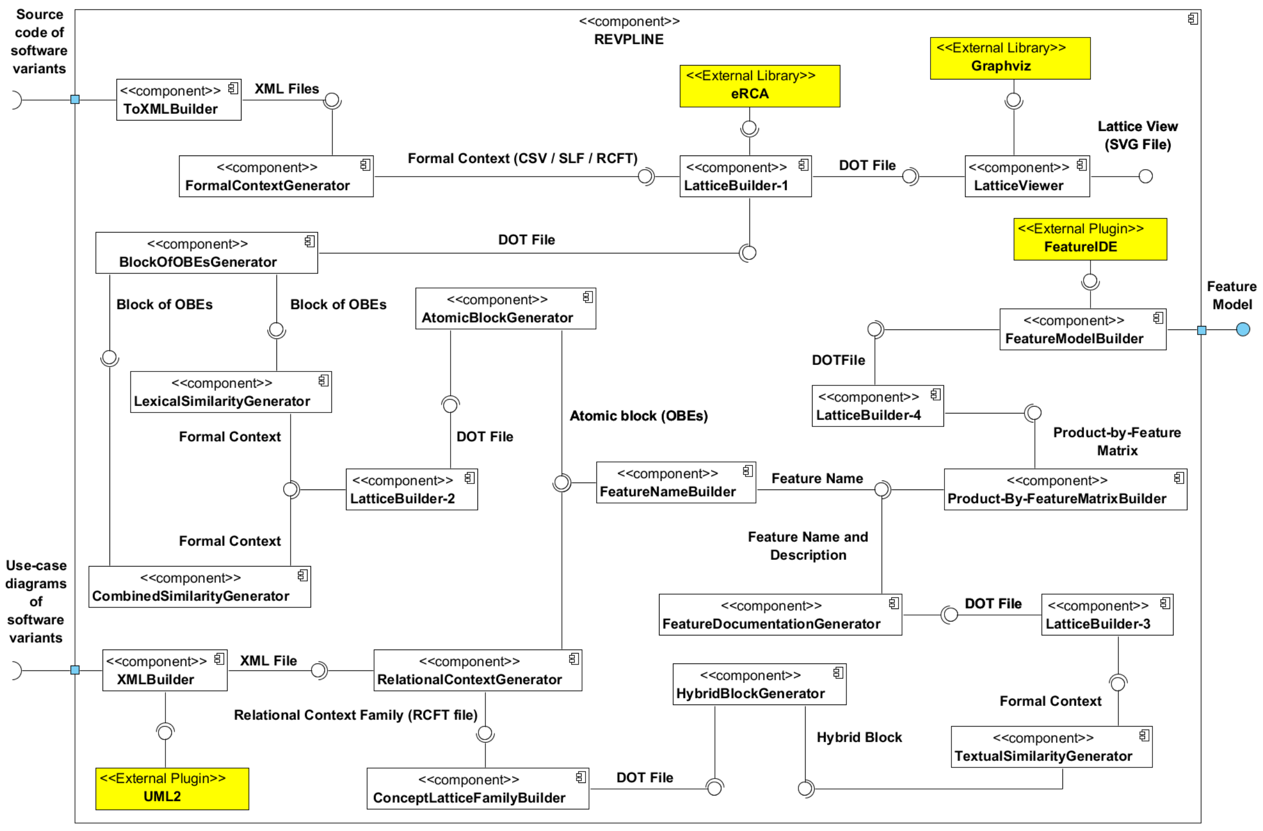

Source code of software variants
<<component>> REVPLINE
<<component>> ToXMLBuilder
XML Files
<<External Library>> eRCA
<<External Library>> Graphviz
<<component>> FormalContextGenerator
Formal Context (CSV / SLF / RCFT)
<<component>> LatticeBuilder-1
DOT File
<<component>> LatticeViewer
Lattice View (SVG File)
<<component>> BlockOfOBEsGenerator
DOT File
<<External Plugin>> FeatureIDE
Block of OBEs
Block of OBEs
<<component>> AtomicBlockGenerator
Feature Model
<<component>> FeatureModelBuilder
DOTFile
<<component>> LexicalSimilarityGenerator
<<component>> LatticeBuilder-4
Atomic block (OBEs)
Formal Context
DOT File
Product-by-Feature Matrix
<<component>> LatticeBuilder-2
<<component>> FeatureNameBuilder
Feature Name
<<component>> Product-By-FeatureMatrixBuilder
Formal Context
Feature Name and Description
Use-case diagrams of software variants
<<component>> CombinedSimilarityGenerator
<<component>> FeatureDocumentationGenerator
DOT File
<<component>> LatticeBuilder-3
<<component>> XMLBuilder
XML File
<<component>> RelationalContextGenerator
<<component>> HybridBlockGenerator
Formal Context
Relational Context Family (RCFT file)
Hybrid Block
<<component>> TextualSimilarityGenerator
<<External Plugin>> UML2
<<component>> ConceptLatticeFamilyBuilder
DOT File

Figure A.1 : A simplified structural view of the architecture of REVPLINE.

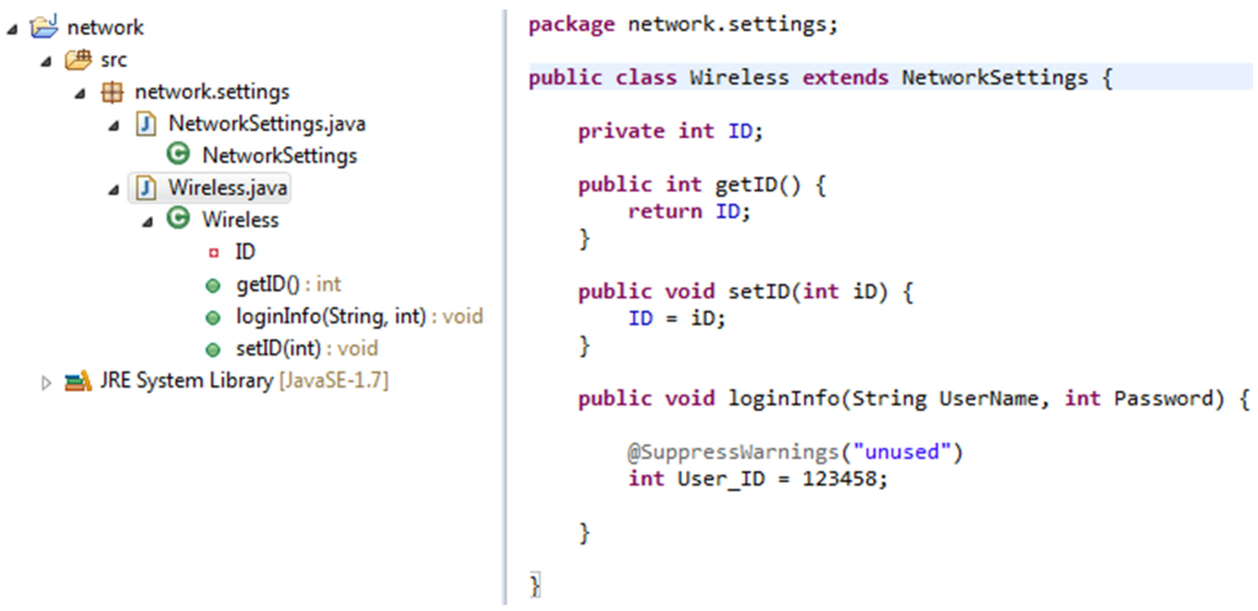
A.2 REVPLINE Components

In this section, we present all components that are used in REVPLINE tool. We provide more details about these components. In addition, we provide some examples according to the component importance. We used some tools to give a general picture and visualization about REVPLINE implementation.

❏ **The ToXMLBuilder Component.** The *ToXMLBuilder* component generates an XML file for each product. This XML file contains all OBEs for the given software. Moreover the XML file contains structural information between OBEs (*e.g. loginInfo* method is inside the *Wireless* class). The

[1] REVPLINE source code : `https://code.google.com/p/revpline-approach/`

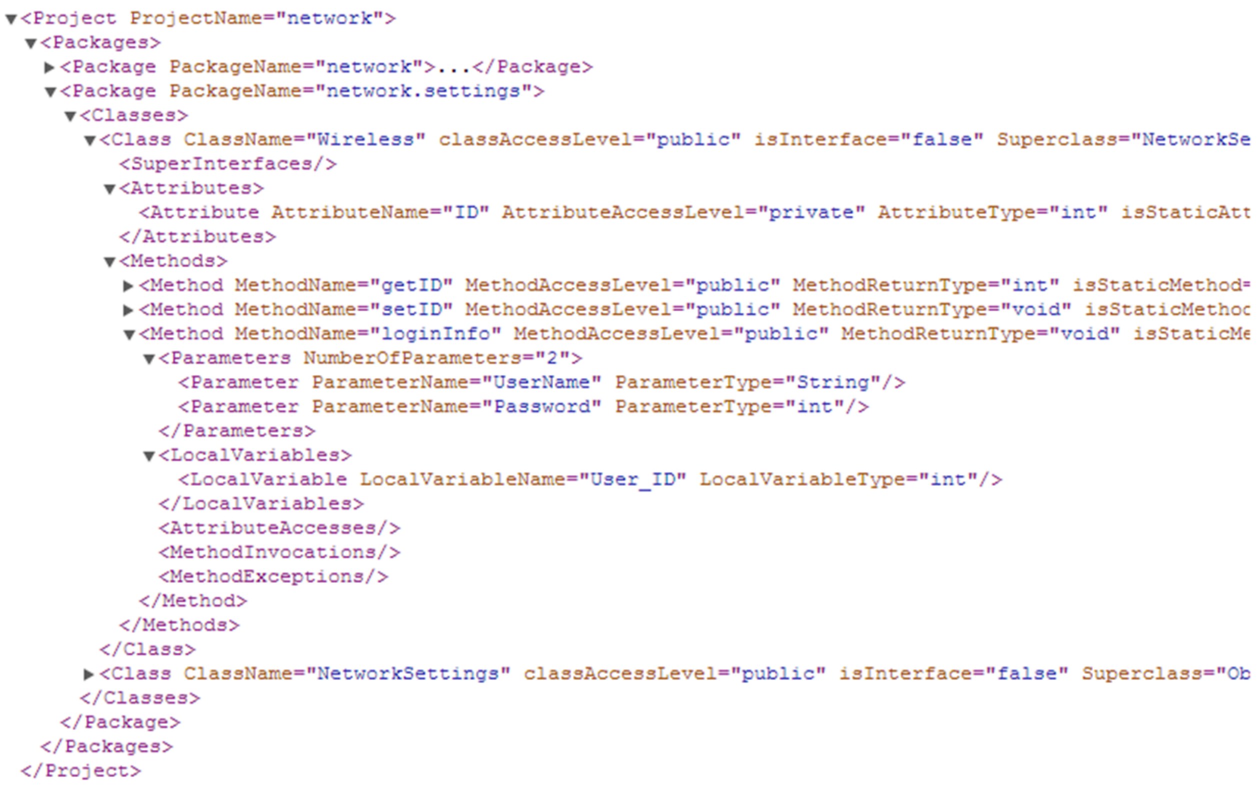
Eclipse JDT (Java Development Tools) and the Eclipse AST (Abstract Syntax Tree) can be used to access, modify and read the elements of a Java program. ASTs are broadly used in numerous fields of software engineering. AST is used as a representation of source code [Rakic and Budimac, 2013]. Figure A.2 shows the Java source code of *network* application. We rely on the AST parser to parse this Java program as an XML file. Figure A.3 shows the extracted XML file.



```
network
  src
    network.settings
      NetworkSettings.java
        NetworkSettings
      Wireless.java
        Wireless
          ID
          getID() : int
          loginInfo(String, int) : void
          setID(int) : void
  JRE System Library [JavaSE-1.7]
```

```java
package network.settings;

public class Wireless extends NetworkSettings {

    private int ID;

    public int getID() {
        return ID;
    }

    public void setID(int iD) {
        ID = iD;
    }

    public void loginInfo(String UserName, int Password) {

        @SuppressWarnings("unused")
        int User_ID = 123458;

    }

}
```

Figure A.2 : Simple Java program "network application".



```xml
▼<Project ProjectName="network">
  ▼<Packages>
    ▶<Package PackageName="network">...</Package>
    ▼<Package PackageName="network.settings">
      ▼<Classes>
        ▼<Class ClassName="Wireless" classAccessLevel="public" isInterface="false" Superclass="NetworkSe
            <SuperInterfaces/>
          ▼<Attributes>
              <Attribute AttributeName="ID" AttributeAccessLevel="private" AttributeType="int" isStaticAtt
            </Attributes>
          ▼<Methods>
            ▶<Method MethodName="getID" MethodAccessLevel="public" MethodReturnType="int" isStaticMethod=
            ▶<Method MethodName="setID" MethodAccessLevel="public" MethodReturnType="void" isStaticMethoc
            ▼<Method MethodName="loginInfo" MethodAccessLevel="public" MethodReturnType="void" isStaticMe
              ▼<Parameters NumberOfParameters="2">
                  <Parameter ParameterName="UserName" ParameterType="String"/>
                  <Parameter ParameterName="Password" ParameterType="int"/>
                </Parameters>
              ▼<LocalVariables>
                  <LocalVariable LocalVariableName="User_ID" LocalVariableType="int"/>
                </LocalVariables>
                <AttributeAccesses/>
                <MethodInvocations/>
                <MethodExceptions/>
              </Method>
            </Methods>
          </Class>
        ▶<Class ClassName="NetworkSettings" classAccessLevel="public" isInterface="false" Superclass="Ob
        </Classes>
      </Package>
    </Packages>
  </Project>
```

Figure A.3 : The OBEs of network application as XML file.

In REVPLINE we rely on the AST parser to extract all source code elements (*i.e.* OBEs) from each variant. The XML file in Figure A.3 represents all OBEs of network application (*e.g.* package "network.settings", class "Wireless", class "NetworkSettings").

❏ **The FormalContextGenerator Component.** This component accepts as input a set of XML files. Based on the input XML files, this component generates the formal context (*cf.* Listing A.1). In this formal context, *objects* are product variants (*e.g.* network-1, network-2) and *attributes* are OBEs (*e.g.* class "Wireless", class "NetworkSettings"). This table is serialized as a *CSV* (comma separated values) file. The CSV is an open computer format representing tabular data values separated by commas.

The formal context table also can serialized as an *RCFT* file. The RCFT file format has been designed to store relational context families (in addition to the formal context). It is somehow similar to the CSV format, but uses | as a separator (*cf.* Listing A.4). In RCFT you can define either formal or relational contexts, while CSV file defines only the formal context.

```
           , package (network), class (Wireless, network), class (Bluetooth, network), ...
Network-1  , x                  , x                        ,                           , ...
Network-2  , x                  , x                        , x                         , ...
Network-3  , x                  , x                        ,                           , ...
Network-4  , x                  , x                        , x                         , ...
```

LISTING A.1 : Formal context describing software variants by their OBEs.

❏ **The LatticeBuilder-1 Component.** Starting from the formal context table, this component builds the concept lattice or the AOC-poset for each formal context. For doing so, the LatticeBuilder-1 component uses an external library, called eRCA[2] to produce DOT[3] file containing the concept lattices of these software variants. DOT is a plain text graph description language (*cf.* Listing A.6). The eRCA is an implementation of the FCA and RCA algorithms. This component accepts as input the formal context and produces the concept lattice as a DOT file. Listings A.4 and A.3 show an example of formal context in different formats.

❏ **The LatticeViewer Component.** This component aims to visualize the concept lattice or the concept lattice family. This component accepts as input the DOT file and produces as output the *AOC-poset*, *concept lattice* or the *concept lattice family* in many formats such as JPG, PNG, EPS, GIF, SVG, PDF. Figure A.7 shows an example of the visualization of a concept lattice family. Figure A.4 shows the visualization of the AOC-poset for the formal context of Listing A.3. Figure A.5 shows the visualization of the concept lattice for the formal context of Listing A.3.

The LatticeViewer component represents the concept lattices as a graph using the DOT file. We will mainly use scalable vector graphics (SVG). SVG is an XML-based file format for describing vector graphics. For doing so, visualization component uses an external library, called Graphviz[4] to produce SVG file. Graphviz is open source graph visualization software. The SVG file can be modified in graphical editors.

[2] https://code.google.com/p/erca/
[3] http://www.graphviz.org/content/dot-language
[4] http://www.graphviz.org/

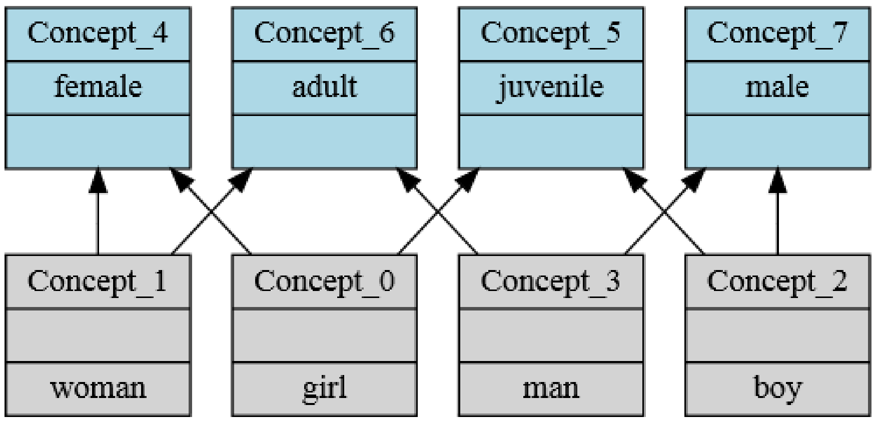


Figure A.4 : The AOC-poset for the formal context of Listing A.3.

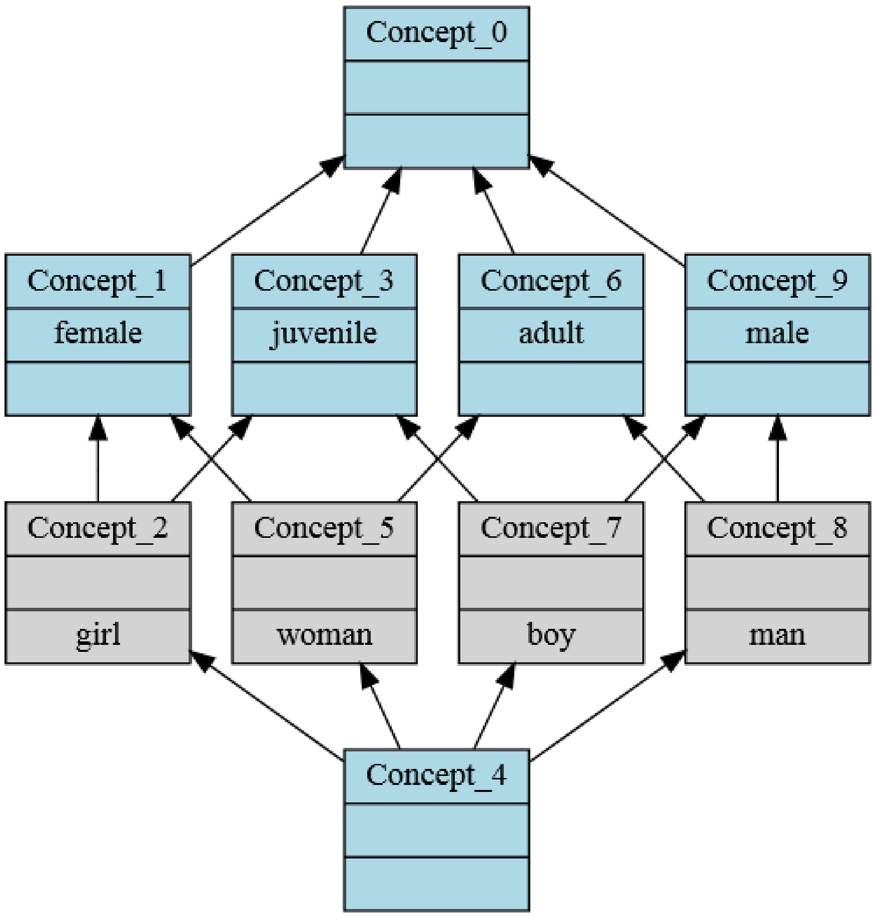


Figure A.5 : The concept lattice for the formal context of Listing A.3.

❑ **The BlockOfOBEsGenerator Component.** This component identifies the common block of OBEs and a set of blocks of variation. This component accepts as input the DOT file which represents the AOC-poset. Then it produces as output the common block and block of variations by parsing the DOT file.

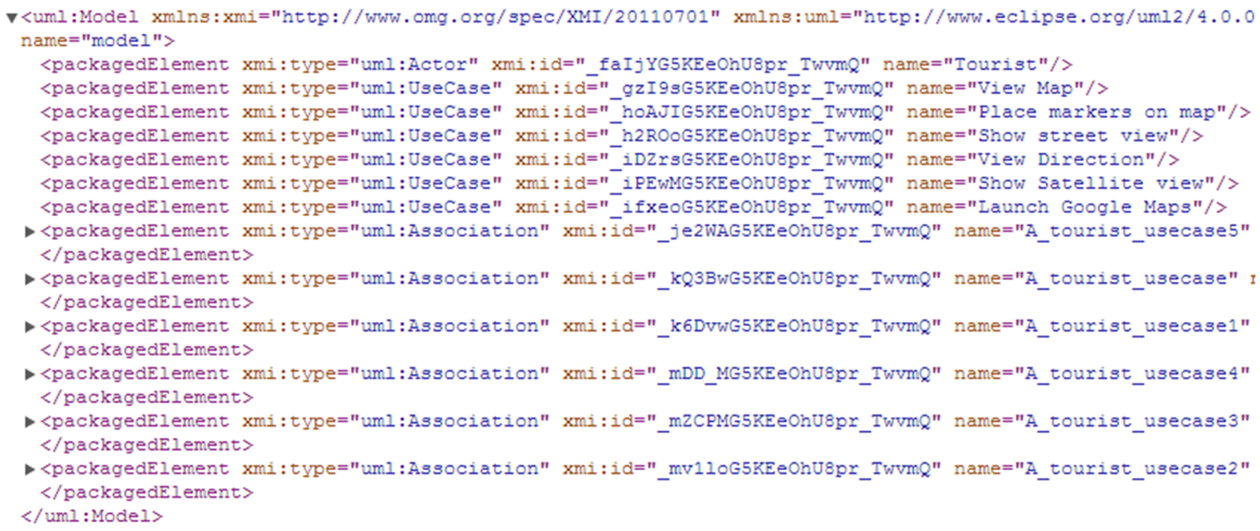
❏ **The LexicalSimilarityGenerator Component.** This component uses to measure the lexical similarity between OBEs of each block (common block or blocks of variation). This component accepts as input a block of OBEs and produces as outputs the lexical similarity matrix (numerical values) in addition to the formal context that describes this similarity.

❏ **The CombinedSimilarityGenerator Component.** This component uses to measure both lexical and structural similarity between OBEs of the given block. This component accepts as input a set of blocks and produces the combined matrix of each block which represents the structural and lexical similarity as a formal context.

❏ **The LatticeBuilder-2 Component.** This component produces the atomic blocks (*i.e.* feature implementations) based on the given formal context representing similarity. This component accepts as input a formal context and produces as output a set of clusters: each one represents an atomic block. To identify the atomic blocks based on the given formal context, this component uses again eRCA to produce the DOT file.

❏ **The AtomicBlockGenerator Component.** This component identifies the atomic blocks based on the entrance DOT file. In other words, this component uses to parse DOT file and identify each atomic block as a set of OBEs.

❏ **The XMLBuilder Component.** This component allows the expert to draw the use-case diagrams of software variants and save them as an XML file. In doing so, this component uses an external eclipse plugin, called UML2 plugin to produce XML file. Figure A.6 shows an example of use-case diagram as an XML file corresponding to the use-case diagram of Figure A.8.



```
▼<uml:Model xmlns:xmi="http://www.omg.org/spec/XMI/20110701" xmlns:uml="http://www.eclipse.org/uml2/4.0.0
 name="model">
   <packagedElement xmi:type="uml:Actor" xmi:id="_faIjYG5KEeOhU8pr_TwvmQ" name="Tourist"/>
   <packagedElement xmi:type="uml:UseCase" xmi:id="_gzI9sG5KEeOhU8pr_TwvmQ" name="View Map"/>
   <packagedElement xmi:type="uml:UseCase" xmi:id="_hoAJIG5KEeOhU8pr_TwvmQ" name="Place markers on map"/>
   <packagedElement xmi:type="uml:UseCase" xmi:id="_h2ROoG5KEeOhU8pr_TwvmQ" name="Show street view"/>
   <packagedElement xmi:type="uml:UseCase" xmi:id="_iDZrsG5KEeOhU8pr_TwvmQ" name="View Direction"/>
   <packagedElement xmi:type="uml:UseCase" xmi:id="_iPEwMG5KEeOhU8pr_TwvmQ" name="Show Satellite view"/>
   <packagedElement xmi:type="uml:UseCase" xmi:id="_ifxeoG5KEeOhU8pr_TwvmQ" name="Launch Google Maps"/>
  ▶<packagedElement xmi:type="uml:Association" xmi:id="_je2WAG5KEeOhU8pr_TwvmQ" name="A_tourist_usecase5"
   </packagedElement>
  ▶<packagedElement xmi:type="uml:Association" xmi:id="_kQ3BwG5KEeOhU8pr_TwvmQ" name="A_tourist_usecase" r
   </packagedElement>
  ▶<packagedElement xmi:type="uml:Association" xmi:id="_k6DvwG5KEeOhU8pr_TwvmQ" name="A_tourist_usecase1"
   </packagedElement>
  ▶<packagedElement xmi:type="uml:Association" xmi:id="_mDD_MG5KEeOhU8pr_TwvmQ" name="A_tourist_usecase4"
   </packagedElement>
  ▶<packagedElement xmi:type="uml:Association" xmi:id="_mZCPMG5KEeOhU8pr_TwvmQ" name="A_tourist_usecase3"
   </packagedElement>
  ▶<packagedElement xmi:type="uml:Association" xmi:id="_mv1loG5KEeOhU8pr_TwvmQ" name="A_tourist_usecase2"
   </packagedElement>
 </uml:Model>
```

Figure A.6 : Use-case diagram as XML file.

❏ **The RelationalContextGenerator Component.** The goal of this component is to generate the relational context family for the feature documentation process. This component accepts as input atomic blocks of OBEs (feature implementations) and the XML files of use-case diagram and produces as output the relational context family based on software configurations. This component produces the relational context family as a single RCFT File. Listing A.5 shows an example of relational context family.

❏ **The ConceptLatticeFamilyBuilder Component.** This component accepts as input the relational context family and produces as output the concept lattice family as a DOT file. This component plays an important to reduce the search space of the feature documentation process by exploiting commonality and variability across software variants. To build the concept lattice family based on the relational context family, this component uses the external library eRCA.

❏ **The HybridBlockGenerator Component.** This component accepts as input the DOT file which contains the concept lattice family which has been generated based on the relational context family. This component collects use-cases and feature implementations into a set of clusters (hybrid blocks) based on the concept lattice family. This component explores the concept lattice family and filters it to produce a set of hybrid blocks from bottom to top.

❏ **The TextualSimilarityGenerator Component.** This component uses to measure the textual (lexical) similarity between use-cases and feature implementations of each hybrid block. This component accepts as input the hybrid block which consists of a collection of use-cases and feature implementations. This component produces as output the lexical similarity matrix (numerical values) in addition to the formal context that describes this similarity.

❏ **The LatticeBuilder-3 Component.** This component accepts as input the formal context which describes the textual similarity between use-cases and feature implementations for each hybrid block. This component produces the concept lattice for the given formal context. This component uses the external library eRCA to produce the DOT file which represents the feature documentation.

❏ **The FeatureDocumentationGenerator Component.** This component identifies the feature documentation by parsing the DOT file (concept lattice). In other words, this component uses to parse DOT file and extract the feature documentation for each implementation based on the use-case name and description.

❏ **The FeatureNameBuilder Component.** This component identifies the feature name based on the OBE names. When the use-case diagrams of software variants are missing, we must use another way to document the mined feature implementation. This component uses to document the mined feature implementation directly from the identifier names of each atomic block.

❏ **The Product-By-FeatureMatrixBuilder Component.** This component uses to build the product-by-feature matrix which contains the mined and documented features from software variants. REVPLINE aims to identify the feature model that describes software variant features. For doing so, this component builds the product-by-feature matrix as a formal context table. This table is serialized as RCFT file (*cf.* Listing A.2).

```
|           | Feature A | Feature B | Feature C | Feature D | Feature E | ...
|Software-1 |     x     |    x      |           |     x     |           | ...
|Software-2 |     x     |    x      |     x     |     x     |    x      | ...
|Software-3 |     x     |    x      |           |     x     |           | ...
```

LISTING A.2 : Formal context describing software variants by their features.

❏ **The LatticeBuilder-4 Component.** This component accepts as input the formal context which describes software configurations (the product-by-feature matrix). This component produces the AOC-poset for the given formal context. This component uses the external library eRCA to produce the AOC-poset as DOT file.

❏ **The FeatureModelBuilder Component.** The feature model builder component uses to parse the DOT file and extract the feature model from the product configurations. The resulting feature models covers the given formal context (*i.e.* product-by-feature matrix). This component builds the FM based on the extracted software configurations. Feature model component uses an external plugin, called FeatureIDE [Thüm *et al.*, 2014] to represent the extracted FM. By parsing the DOT file we get the FM with its constraints as an XML file and other format such as SXFM file. SXFM is for Simple XML Feature Model format. SXFM format is used in SPLOT homepage[5]. REVPLINE also produces the FM as DOT file[6].

A.3 External Libraries

In REVPLINE we use two external libraries: eRCA and Graphviz. We describe each library. Moreover, we explain the role of each library in REVPLINE. We provide some illustrative examples to explain these libraries.

A.3.1 eRCA

REVPLINE uses eRCA as external library. eRCA is a framework that eases the use of Formal and Relational Concept Analysis, a neat clustering technique. eRCA provides easy import of formal contexts from CSV files (*resp.* relational contexts from RCFT files). eRCA also provides easy export of the produced lattices to DOT graphs (that can be later on exported to JPG, PNG, EPS and SVG using the Graphviz library).

For FCA, the formal context must be built as CSV or RCFT files. After that eRCA uses these files as inputs and generates the lattices as DOT file. For RCA, the relational context family must be built as RCFT files. After that eRCA uses these files as inputs and generates the concept lattice family as DOT file.

[5]http://gsd.uwaterloo.ca:8088/SPLOT/index.html
[6]http://www.lirmm.fr/~seriai/encadrements/theses/rafat/index.php?n=T.FMD

```
      , female, juvenile, adult, male,
girl  , x     , x       ,      ,     ,
woman , x     ,         , x    ,     ,
boy   ,       , x       ,      , x   ,
man   ,       ,         , x    , x   ,
```

LISTING A.3 : Formal context as CSV File.

```
FormalContext Human
|      |female | juvenile | adult  | male   |
|girl  | X     |  X       |        |        |
|woman | X     |          |   X    |        |
|boy   |       |  X       |        |   X    |
|man   |       |          |   X    |   X    |
```

LISTING A.4 : Formal context as RCFT File.

Listing A.3 and Listing A.4 show the same formal context in two formats: CSV and RCFT. In the first one, we represent the formal context as a CSV file. In the second example, we represent the same formal context as RCFT file. This formal context has been taken from the web page of Uta Priss[7].

Listing A.5 shows an example of relational context family. It contains two formal contexts (Animals and Places) and one relational context (Lives).

```

FormalContext Animals
|         |
| eagle   |
| bat     |
| catfish |

FormalContext Places
|          |
| mountain |
| cave     |
| river    |

RelationalContext lives
source Animals
target Places
scaling com.googlecode.erca.framework.algo.scaling.Wide
|         | mountain |  cave   | river |
| eagle   | x        |         |       |
| bat     |          |   x     |       |
| catfish |          |         |   x   |
```

LISTING A.5 : The relational context family for animals as RCFT File.

In REVPLINE we rely five times in eRCA. Firstly, we rely on eRCA to extract common block and blocks of variation. Secondly, to cluster OBEs based on the lexical/structural similarity into atomic blocks. Thirdly, to cluster use-cases and feature implementations into hybrid blocks. Fourthly, to cluster feature name and implementation of the hybrid block based on the lexical similarity. Fifthly, to extract the AOC-poset which represents software configurations in order to extract an FM.

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

A.3.2 Graphviz

Graphviz is open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. It has important applications in networking, software engineering, database and web design, and in machine learning. The Graphviz layout programs take descriptions of graphs in a simple text language (*e.g.* DOT), and make diagrams in useful formats, such as images and SVG for web pages, PDF or Postscript for inclusion in other documents, or display in an interactive graph browser. Graphviz has many useful features for concrete diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyper-links, and custom shapes. The DOT file is the perfect format if we deal with directed graphs. DOT draws directed graphs as hierarchies[8].

```
digraph G {
rankdir=BT;
subgraph cluster_Motion {
label = Motion;
2126858590 [shape=record,style=filled,fillcolor=orange,label="{Concept_0||fly\nswim\n}"];}
subgraph cluster_Animals {
label = Animals;
1247017815 [shape=record,style=filled,fillcolor=orange,label="{Concept_1|lives :
Concept_3\nmoves : Concept_0\n|eagle\nbat\ncatfish\n}"];
1335505632 [shape=record,style=filled,fillcolor=lightblue,label="{Concept_2|\n|}"];
   1335505632 -> 1247017815}
subgraph cluster_Places {
label = Places;
344078580 [shape=record,style=filled,fillcolor=orange,label="{Concept_3|offers :
Concept_5\n|mountain\ncave\nriver\n}"];
1902716336 [shape=record,style=filled,fillcolor=lightblue,label="{Concept_4|\n|}"];
   1902716336 -> 344078580}
subgraph cluster_Food {
label = Food;
774471675 [shape=record,style=filled,fillcolor=orange,label="{Concept_5|eatBy :
Concept_1\n|mouse\ninsect\nfish\n}"];
647057258 [shape=record,style=filled,fillcolor=lightblue,label="{Concept_6|\n|}"];
   647057258 -> 774471675
} }
```

LISTING A.6 : The concept lattice family of Listing A.5 as DOT file.

Listing A.6 shows an example of DOT file. This DOT file was produced by eRCA external library. In REVPLINE, the DOT file contains the most important information. We rely on the DOT parser to get the common block of OBEs and blocks of variations. In addition, we rely on the DOT file to get atomic blocks of OBEs (*i.e.* feature implementations). Also DOT file provides us with feature name and description. Finally, we rely on the DOT file to extract the FM by parsing the DOT file. From the DOT file we get the concept lattice and the concept lattice family as a graph. Figure A.7 shows the graph obtained from the DOT file in the previous example.

[8]http://www.graphviz.org/pdf/dotguide.pdf

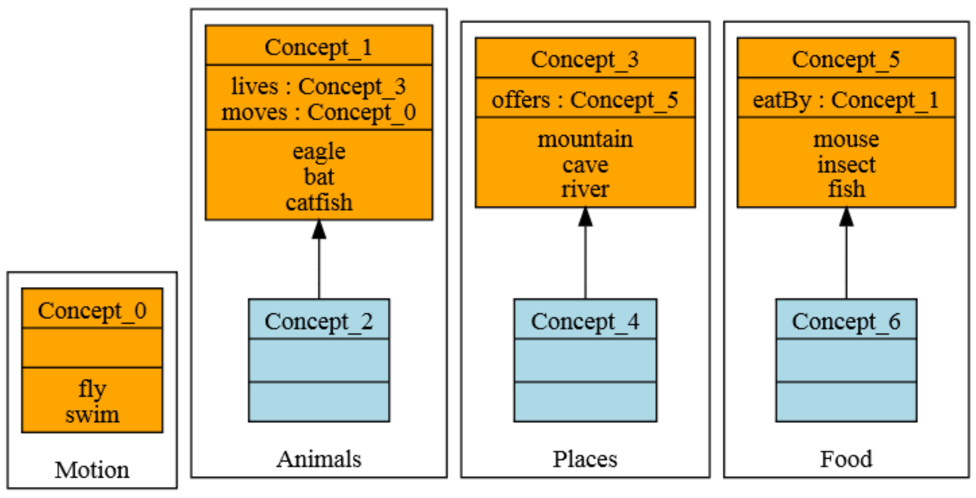


Figure A.7 : The SVG file of the DOT file of Listing A.6.

A.4 Eclipse Plugins

The Eclipse Modelling Framework[9] (EMF) is a facility provided by the Eclipse IDE to implement modelling languages and generate tools to manipulate instances of those languages from programs. We use in our tool two Eclipse plugins based on EMF: UML2[10] and FeatureIDE[11]. UML2 is an implementation of UML and therefore allows us to draw, load, modify and save use-case diagrams. FeatureIDE is an Eclipse plugin for Feature-Oriented Software Development. We describe each plugin in addition to explain the role of these plugins in REVPLINE approach.

A.4.1 UML2 Plugin

The Unified Modeling Language (UML) is a visual language for capturing software designs and patterns. The first version of UML was defined in 1994 and released by the Object Management Group (OMG) in 1997 as UML v.1.1. The syntax and a semantics of UML are defined by the OMG [Ambler, 2004]. UML2 is an EMF-based implementation of the Unified Modeling Language (UML^{TM}) 2.x OMG metamodel for the Eclipse platform. The basic building block for UML is a diagram. UML divides diagrams into structural diagrams and behavioral diagrams. In our work, we use UML2 as external plugin. UML2 allows us to draw and save the use-case diagrams of software variants (*cf.* Figure A.8).

[9] http://projects.eclipse.org/projects/modeling.emf
[10] http://www.eclipse.org/modeling/mdt/?project=uml2#uml2
[11] http://www.fosd.de/fide

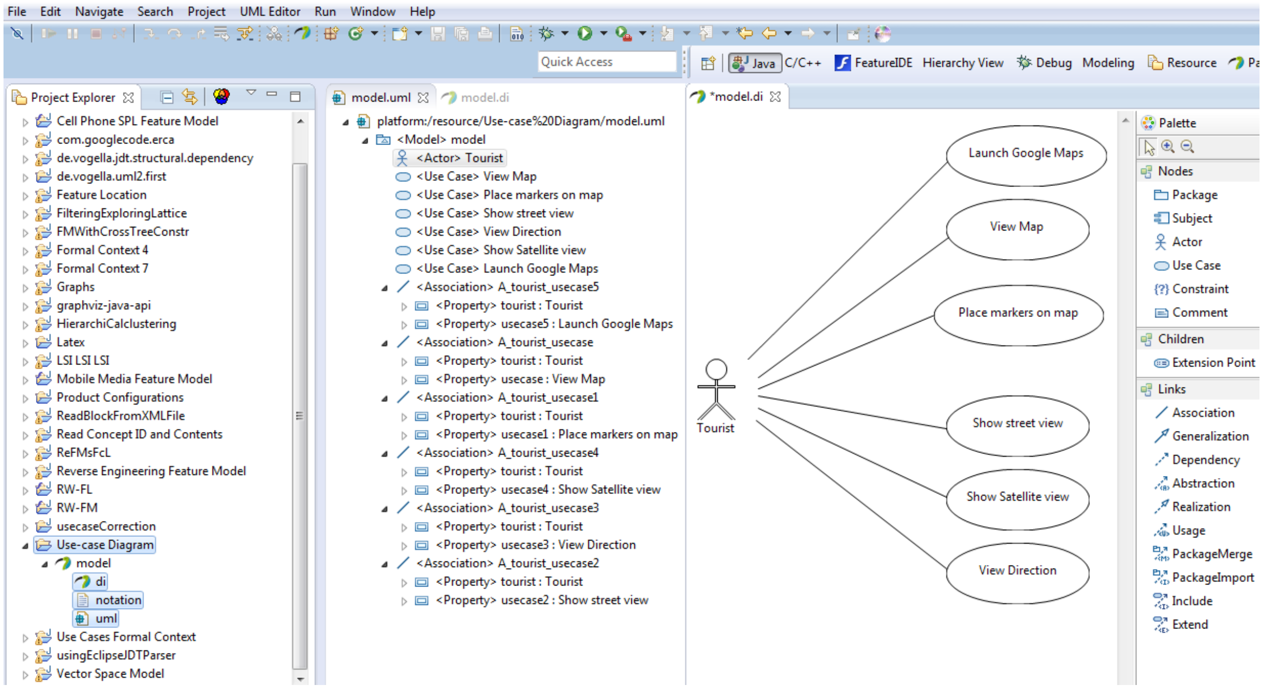

File Edit Navigate Search Project UML Editor Run Window Help
Quick Access
Java C/C++ FeatureIDE Hierarchy View Debug Modeling Resource
Project Explorer
Cell Phone SPL Feature Model
com.googlecode.erca
de.vogella.jdt.structural.dependency
de.vogella.uml2.first
Feature Location
FilteringExploringLattice
FMWithCrossTreeConstr
Formal Context 4
Formal Context 7
Graphs
graphviz-java-api
HierarchiClalclustering
Latex
LSI LSI LSI
Mobile Media Feature Model
Product Configurations
ReadBlockFromXMLFile
Read Concept ID and Contents
ReFMsFcL
Reverse Engineering Feature Model
RW-FL
RW-FM
usecaseCorrection
Use-case Diagram
model
di
notation
uml
Use Cases Formal Context
usingEclipseJDTParser
Vector Space Model
model.uml
model.di
platform:/resource/Use-case%20Diagram/model.uml
<Model> model
<Actor> Tourist
<Use Case> View Map
<Use Case> Place markers on map
<Use Case> Show street view
<Use Case> View Direction
<Use Case> Show Satellite view
<Use Case> Launch Google Maps
<Association> A_tourist_usecase5
<Property> tourist : Tourist
<Property> usecase5 : Launch Google Maps
<Association> A_tourist_usecase
<Property> tourist : Tourist
<Property> usecase : View Map
<Association> A_tourist_usecase1
<Property> tourist : Tourist
<Property> usecase1 : Place markers on map
<Association> A_tourist_usecase4
<Property> tourist : Tourist
<Property> usecase4 : Show Satellite view
<Association> A_tourist_usecase3
<Property> tourist : Tourist
<Property> usecase3 : View Direction
<Association> A_tourist_usecase2
<Property> tourist : Tourist
<Property> usecase2 : Show street view
*model.di
Launch Google Maps
View Map
Place markers on map
Tourist
Show street view
Show Satellite view
View Direction
Palette
Nodes
Package
Subject
Actor
Use Case
Constraint
Comment
Children
Extension Point
Links
Association
Generalization
Dependency
Abstraction
Realization
Usage
PackageMerge
PackageImport
Include
Extend

Figure A.8 : UML2 plugin for Eclipse Juno.

A.4.2 FeatureIDE Plugin

FeatureIDE is an open-source framework for feature-oriented software development (FOSD) based on Eclipse. FOSD is a paradigm for construction, customization, and synthesis of software systems. FeatureIDE supports all phases of feature-oriented software development for the development of SPLs: domain analysis, domain implementation, requirements analysis, and software generation. Different SPL implementation techniques are integrated such as feature-oriented programming (FOP), aspect-oriented programming (AOP), delta-oriented programming (DOP), and preprocessors [Thüm *et al.*, 2014]. In REVPLINE, the FeatureIDE plugin uses to represent the mined FM with its constraints (*cf.* Figure A.9).

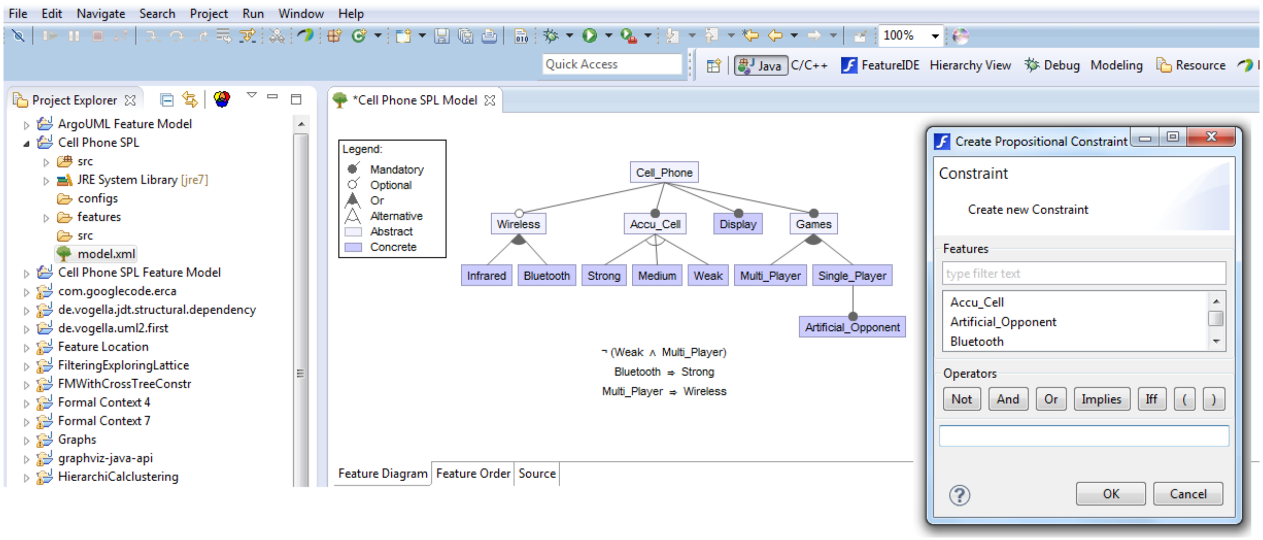


Figure A.9 : The graphical feature model editor of FeatureIDE.

A.5 Statistical information regarding REVPLINE source code

In this section, we present the statistical information regarding REVPLINE source code such as the number of packages, classes, methods and so on. We rely on the Moose[12] technology to get all needed information regarding the source code. Moose is a generic platform for engineers that want to understand data in general and software systems in particular. We rely on the VerveineJ[13] parser to export the MSE[14] file. But what exactly is MSE? MSE is the default file format supported by Moose. It is a generic file format and can describe any model. It is similar to XML, the main difference being that instead of using verbose tags, it makes use of parentheses to denote the beginning and ending of an element.

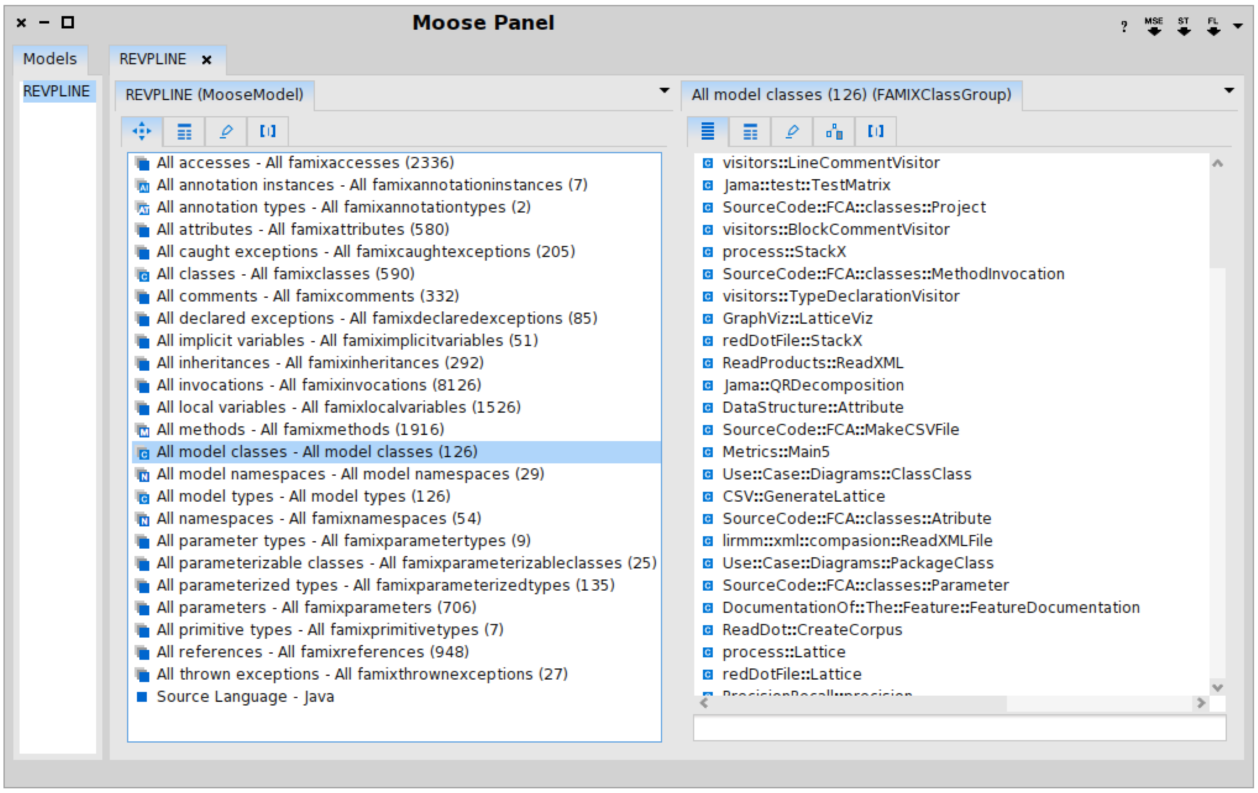
```
( (FAMIX.Package (id: 202)
    (name 'anotherPackage')
    (parentPackage (ref: 201)))
  (FAMIX.Class (id: 2)
    (name 'ClassA')
    (container (ref: 1))
    (parentPackage (ref: 201)))
  (FAMIX.Method
    (name 'methodA1')
    (signature 'methodA1()')
    (parentType (ref: 2))
    (LOC 2))
  (FAMIX.Attribute
    (name 'attributeA1')
    (parentType (ref: 2)))
  (FAMIX.Inheritance
    (subclass (ref: 3))
    (superclass (ref: 2))))
```

LISTING A.7 : Snippet provides an example of a small model of the MSE file.

Listing A.7 shows an example[15] of a small model of the MSE file. Figure A.10 shows the Moose[16] Finder open on a model. From this Figure we can see some statistical information regarding the source code of REVPLINE approach. The interested reader can find more information about our use of Moose, MSE, and VerveineJ in our website[17] (simple tutorial).

We also use the CodeCity[18] (visualize software system as a city) software to visualize REVPLINE approach. CodeCity is an integrated environment for software analysis, in which software systems are visualized as interactive, navigable 3D cities (*cf.* Figure A.11). The classes are represented as buildings in the city, while the packages are depicted as the districts in which the buildings reside. The visible properties of the city artifacts depict a set of chosen software metrics. CodeCity accepts the MSE file only as input. To use CodeCity software the MSE file must be in FAMIX 2.1 format, otherwise it won't work. FAMIX is a family of meta-models[19]. The Core of FAMIX is a language independent meta-model that describes the static structure of object-oriented software systems.

[12] http://moosetechnology.org/
[13] http://www.moosetechnology.org/tools/verveinej
[14] http://www.themoosebook.org/book/internals/petit-parser/mse
[15] http://www.themoosebook.org/book/externals/import-export/mse
[16] Moose download : http://moosetechnology.org/download
[17] http://www.lirmm.fr/~seriai/encadrements/theses/rafat/uploads/T/mse.pdf
[18] http://www.inf.usi.ch/phd/wettel/codecity.html
[19] http://www.moosetechnology.org/docs/famix/3.0



All accesses - All famixaccesses (2336)
All annotation instances - All famixannotationinstances (7)
All annotation types - All famixannotationtypes (2)
All attributes - All famixattributes (580)
All caught exceptions - All famixcaughtexceptions (205)
All classes - All famixclasses (590)
All comments - All famixcomments (332)
All declared exceptions - All famixdeclaredexceptions (85)
All implicit variables - All famiximplicitvariables (51)
All inheritances - All famixinheritances (292)
All invocations - All famixinvocations (8126)
All local variables - All famixlocalvariables (1526)
All methods - All famixmethods (1916)
All model classes - All model classes (126)
All model namespaces - All model namespaces (29)
All model types - All model types (126)
All namespaces - All famixnamespaces (54)
All parameter types - All famixparametertypes (9)
All parameterizable classes - All famixparameterizableclasses (25)
All parameterized types - All famixparameterizedtypes (135)
All parameters - All famixparameters (706)
All primitive types - All famixprimitivetypes (7)
All references - All famixreferences (948)
All thrown exceptions - All famixthrownexceptions (27)
Source Language - Java

visitors::LineCommentVisitor
Jama::test::TestMatrix
SourceCode::FCA::classes::Project
visitors::BlockCommentVisitor
process::StackX
SourceCode::FCA::classes::MethodInvocation
visitors::TypeDeclarationVisitor
GraphViz::LatticeViz
redDotFile::StackX
ReadProducts::ReadXML
Jama::QRDecomposition
DataStructure::Attribute
SourceCode::FCA::MakeCSVFile
Metrics::Main5
Use::Case::Diagrams::ClassClass
CSV::GenerateLattice
SourceCode::FCA::classes::Atribute
lirmm::xml::compasion::ReadXMLFile
Use::Case::Diagrams::PackageClass
SourceCode::FCA::classes::Parameter
DocumentationOf::The::Feature::FeatureDocumentation
ReadDot::CreateCorpus
process::Lattice
redDotFile::Lattice

Figure A.10 : The Moose Finder open on a model.

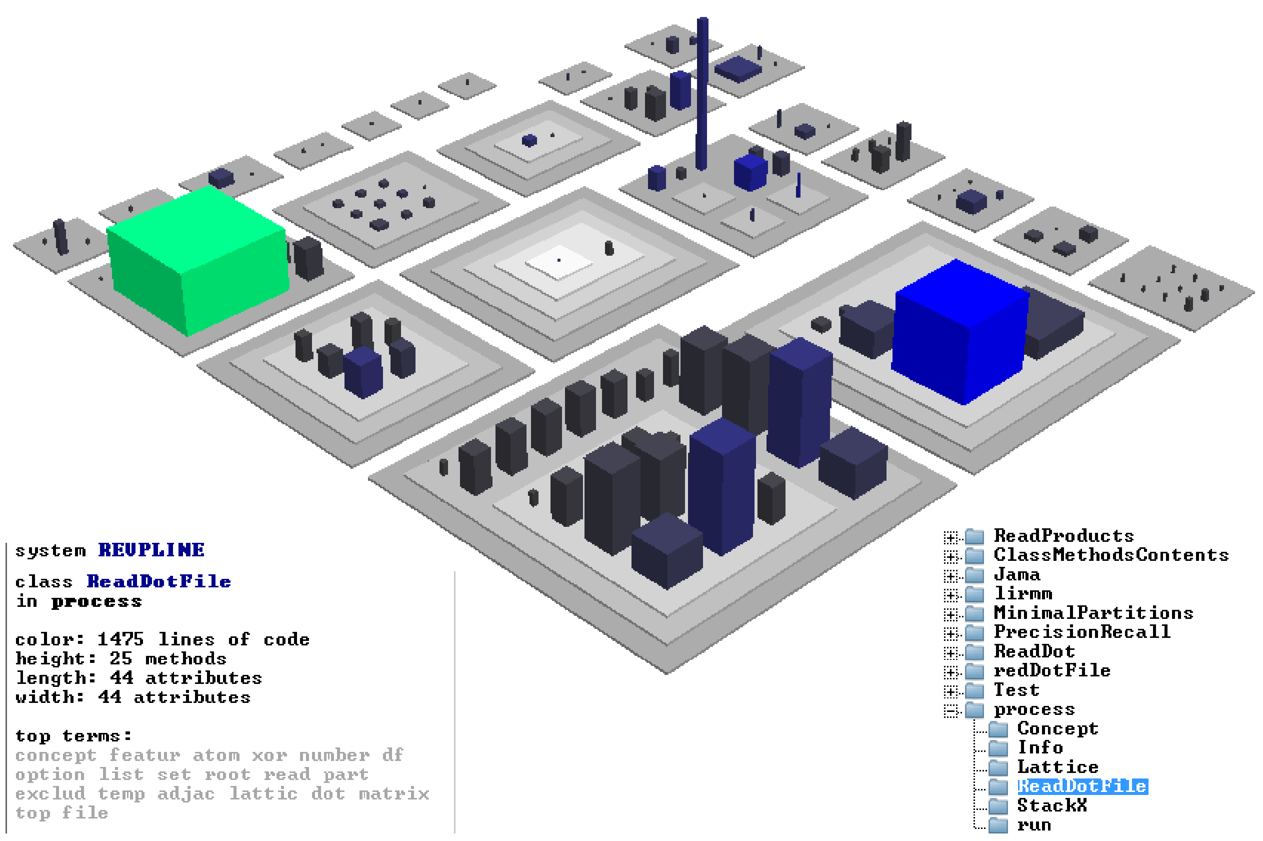
In Figure A.11, we represent classes of REVPLINE as buildings located in districts representing the packages where the classes are defined. The visual properties of the city artifacts reflect metric values. For instance, the *ReadDotFile* class (in the *process* package) has 25 methods (*the building height*), 44 attributes (*building base size*) and 1475 lines of code (*color of the building*, from dark gray "low" to intense blue "high").

Table A.1 shows statistical information regarding REVPLINE implementation. We rely on inFusion software[20] to get this information. More information about REVPLINE implementation is available at our website[21].

Table A.1 : Statistical information regarding REVPLINE source code.

#	Metric Name	Value
1	Total number of lines of code	29,379
2	Number of packages	34
3	Number of classes	143
4	Number of methods	1,115

[20] http://www.intooitus.com/products/infusion
[21] http://www.lirmm.fr/~seriai/encadrements/theses/rafat/index.php?n=T.In



```
system REVPLINE

class ReadDotFile
in process

color: 1475 lines of code
height: 25 methods
length: 44 attributes
width: 44 attributes

top terms:
concept featur atom xor number df
option list set root read part
exclud temp adjac lattic dot matrix
top file
```

Figure A.11 : Representation of a REVPLINE source code in CodeCity.

A.6 Conclusion

In this appendix, we presented the prototype implementation in Java. We also presented the structural view of REVPLINE architecture as a component diagram, then we explained the role of each component in the proposed approach. We presented the external libraries and plugins we used in our prototype. We presented an example regarding some components for better understanding.

List of Figures

2.1 Costs for developing n kinds of systems as single systems compared to SPLE [Clements and Northrop, 2002]. . . . 13
2.2 Overview of an engineering process for software product lines [Apel *et al.*, 2013]. . . . 14
2.3 Cell phone SPL feature model. . . . 17
2.4 The clone-and-own approach. . . . 19
2.5 The concept lattice for the context in Table 2.3 via Galicia. . . . 21
2.6 The concept lattice for the context in Table 2.3 via Con Exp. . . . 22
2.7 The concept lattice for the formal context of Table 2.3 via eRCA. . . . 22
2.8 The AOC-poset for the formal context of Table 2.3 via eRCA. . . . 23
2.9 The entities of the Mexican food example. . . . 23
2.10 Concept 85, 73 and 38 of the Restaurant, MexicanDish and Ingredient concept lattice. 25
2.11 Part of the concept lattice family of the Mexican food example. . . . 26
2.12 Graphical representation of the SVD (*resp.* reduced SVD) for TDM. . . . 29

3.1 The basic elements of REVPLINE approach. . . . 69

4.1 Feature location in product variants (inspired by [Xue *et al.*, 2012]). . . . 75
4.2 Object-oriented source code variation. . . . 76
4.3 *OBE to feature* mapping model. . . . 78
4.4 Drawing shapes software variants feature model. . . . 81
4.5 The feature mining process. . . . 82
4.6 The AOC-poset for the formal context of Table 4.2. . . . 83
4.7 The lexical similarity between OBEs of ($Concept$_5) in Figure 4.6 as a directed graph. 86
4.8 The lexical and structural similarity between OBEs of ($Concept$_1) in Figure 4.6 as a directed graph. . . . 90
4.9 Atomic Blocks Mined from ($Concept$_5) in Figure 4.6. . . . 91
4.10 Atomic Blocks Mined from ($Concept$_1) in Figure 4.6. . . . 91
4.11 The mined feature model of drawing shapes software variants. . . . 91

5.1 Use-case diagram of version 1 from mobile media software [Conejero *et al.*, 2012]. . . . 99
5.2 Use-case diagram of version 8 from mobile media software[Conejero *et al.*, 2012]. . . . 100
5.3 The use-case diagrams of the four MTG software variants. . . . 101
5.4 The extended version of *OBE to feature mapping model*. . . . 104
5.5 The feature documentation process. . . . 106
5.6 The common, shared and unique use-cases (*resp.* feature implementations) across software product variants. . . . 107
5.7 The concept lattice family of relational context family in Table 5.3. . . . 108
5.8 Exploring and filtering the hybrid blocks CLF to identify features documentation. . . . 109
5.9 Constructing a raw corpus from hybrid block. . . . 110

5.10 The lexical similarity between use-cases and feature implementations as a directed graph. . . . 112
5.11 The documented features from *Concept_1* in Figure 5.7. . . . 113

6.1 From configurations to a feature model. . . . 118
6.2 Cell phone SPL feature model [Haslinger, 2012]. . . . 119
6.3 FM reverse engineering process. . . . 120
6.4 The AOC-poset for the formal context of Table 6.1. . . . 122
6.5 The mandatory features (*i.e.* base group of features) identified from the cell phone product configurations. . . . 123
6.6 An atomic group of features (AND-group) identified from the cell phone product configurations. . . . 124
6.7 Xor group of features identified from the cell phone product configurations. . . . 126
6.8 Or group of features identified from the cell phone product configurations. . . . 126
6.9 The resulting FM based on the product configurations of Table 6.1. . . . 128
6.10 The mined FM without groups of features (*i.e.* without the abstract features). . . . 130

7.1 ArgoUML-SPL feature model [Couto *et al.*, 2011]. . . . 137
7.2 The use-case diagram of the second release of ArgoUML software variants. . . . 138
7.3 The AOC-poset for the formal context of Table 7.9. . . . 144
7.4 The extracted FM from Table 7.9. . . . 145
7.5 Public health complaint application interface: update employee screen. . . . 146
7.6 Part of the AOC-poset of Health complaint-SPL. . . . 147
7.7 Feature model for the public Health complaint-SPL. . . . 148
7.8 Public Health complaint-SPL use case diagram. . . . 148
7.9 The CLF of the RCF for features documentation of health complaint software variants. 152
7.10 The AOC-poset for the formal context of Table 7.14. . . . 154
7.11 The extracted FM from Table 7.14. . . . 154
7.12 Mobile Media feature model [Tizzei *et al.*, 2011]. . . . 156
7.13 Use case diagram for release 3 of Mobile Media [Conejero *et al.*, 2012]. . . . 156
7.14 The concept lattice family of the relational context family for features documentation. 159
7.15 The AOC-poset for the formal context of Table 7.20. . . . 160
7.16 The extracted FM from Table 7.20. . . . 161
7.17 The mined FM from Mobile Media via FeatureIDE plugin. . . . 162

8.1 Our position in the lattice of related work. . . . 172

A.1 A simplified structural view of the architecture of REVPLINE. . . . 176
A.2 Simple Java program "network application". . . . 177
A.3 The OBEs of network application as XML file. . . . 177
A.4 The AOC-poset for the formal context of Listing A.3. . . . 179
A.5 The concept lattice for the formal context of Listing A.3. . . . 179
A.6 Use-case diagram as XML file. . . . 180
A.7 The SVG file of the DOT file of Listing A.6. . . . 185
A.8 UML2 plugin for Eclipse Juno. . . . 186
A.9 The graphical feature model editor of FeatureIDE. . . . 186
A.10 The Moose Finder open on a model. . . . 188
A.11 Representation of a REVPLINE source code in CodeCity. . . . 189

List of Tables

2.1 Valid product configurations of cell phone SPL. 18
2.2 Animals and their characteristics. 20
2.3 A formal context for animals and their characteristics. 20
2.4 The formal contexts of the Mexican Food RCF. 24
2.5 The relational context family of the Mexican Food RCF. 24
2.6 LSI example: technical memo example. 29
2.7 The 12-term by 9-document matrix. 30
2.8 SVD – The T (term) matrix. 30
2.9 SVD – The singular values (S) matrix. 31
2.10 SVD – The D (document) matrix. 31
2.11 The $TDM = TSD'(T \cdot S \cdot D)$. 31
2.12 The TDM_k matrix. 31
2.13 The term-document matrix and term-query matrix. 33
2.14 The T, S and D matrices. 33
2.15 The T_k, S_k and D_k matrices (TSD'). 33
2.16 The new query vector coordinates in the reduced 2-dimensional space. 34

3.1 Summary of the feature location approaches (the selected papers). 41
3.2 Summary of feature location studies (comparison table). 49
3.3 Summary of source code comprehension studies (the selected papers). 50
3.4 Summary of source code documentation studies (comparison table). 58
3.5 Summary of reverse engineering FMs studies (the selected papers). 59
3.6 Summary of reverse engineering FMs studies (comparison table). 67

4.1 The features of drawing shapes software product variants. 80
4.2 A formal context describing drawing shapes software variants by their features. 83
4.3 The term-document and the term-query matrices of ($Concept\_5$) in Figure 4.6. 85
4.4 The similarity matrix of ($Concept\_5$) in Figure 4.6. 86
4.5 Formal context of ($Concept\_5$) in Figure 4.6. 87
4.6 The dependency structure matrix of ($Concept\_1$) in Figure 4.6. 88
4.7 The similarity matrix of ($Concept\_1$) in Figure 4.6. 89
4.8 The lexical similarity matrix of ($Concept\_1$) in Figure 4.6. 89
4.9 The combined matrix of ($Concept\_1$) in Figure 4.6. 90
4.10 Feature sets of drawing shapes software variants (*i.e.* software configurations). 92
4.11 Summary of REVPLINE: feature location in a collection of software product variants. . . 95

5.1 The use-cases of MTG software variants. 101
5.2 The mined feature implementations from MTG software variants. 102
5.3 The RCF for features documentation. 108

5.4 The term-document and the term-query matrices of *Concept_1* in Figure 5.7. 111
5.5 The cosine similarity matrix of *Concept_1* in Figure 5.7. 112
5.6 Formal context of *Concept_1* in Figure 5.7. 113
5.7 The *product-by-feature matrix* for MTG software variants. 113
5.8 OBE names, tokens, weight and strongest weighted tokens for *show street view* feature implementation. 115
5.9 Summary of REVPLINE: documenting the mined feature implementation. 116

6.1 Valid product configurations of cell phone SPL [Haslinger, 2012]. 119
6.2 All valid product configurations that are defined by the mined FM. 131
6.3 The result of the product configurations that are identified by the mined cell phone FM. 131
6.4 Summary of REVPLINE: reverse engineering FMs from software configurations. . . . 132

7.1 ArgoUML software product variants. 137
7.2 Feature location in a collection of ArgoUML software variants based on lexical similarity. 139
7.3 Size metric and the variability levels of ArgoUML features. 139
7.4 The identified junctions between feature implementations of ArgoUML-SPL. 140
7.5 Feature location in a collection of ArgoUML software variants based on lexical and structural similarity. 140
7.6 Comparing the two ways: lexical versus structural and lexical similarity. 141
7.7 The descriptions of the ArgoUML software product variants. 142
7.8 Features documented from ArgoUML software variants based on use-case diagrams. 143
7.9 The Product-by-feature matrix for ArgoUML software product variants. 143
7.10 Health complaint software product variants. 146
7.11 Features mined from health complaint software variants. 149
7.12 Features documented from Health complaint software product variants. 150
7.13 The relational context for feature documentation of Health complaint software variants. 151
7.14 The Product-by-feature matrix for Health complaint software variants. 153
7.15 Summary of evolution scenarios in Mobile Media. 155
7.16 Mobile Media software product variants. 155
7.17 Features mined from Mobile Media software product variants. 157
7.18 Features documented from Mobile Media software product variants. 158
7.19 The relational context for feature documentation. 159
7.20 The Product-by-feature matrix for Mobile Media software product variants. 160
7.21 The results of the product configurations that are identified by the mined FMs. 163

8.1 The selected criteria used to categorize the selected work with REVPLINE approach. . 171

A.1 Statistical information regarding REVPLINE source code. 188

List of Listings

2.1 Valid product configuration. 17
2.2 Illegal product configuration. 18
2.3 Two examples of synonymy. 27
2.4 An example of polysemy. 27
7.1 Part of the sequence diagram feature implementation. 140
7.2 The *complaint status* feature implementation. 150
7.3 The description of the *view sorted photos* use-case in Figure 7.13. 156
7.4 Part of the favourites feature implementation. 158
7.5 The mined FM as SXFM format. 162
A.1 Formal context describing software variants by their OBEs. 178
A.2 Formal context describing software variants by their features. 182
A.3 Formal context as CSV File. 183
A.4 Formal context as RCFT File. 183
A.5 The relational context family for animals as RCFT File. 183
A.6 The concept lattice family of Listing A.5 as DOT file. 184
A.7 Snippet provides an example of a small model of the MSE file. 187

Bibliography

[Acher *et al.*, 2011] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, et Philippe Lahire. Reverse engineering architectural feature models. In *Proceedings of the 5th European conference on Software architecture*, ECSA'11, pages 220–235, Berlin, Heidelberg, 2011. Springer-Verlag.

[Acher *et al.*, 2012] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, et Philippe Lahire. On extracting feature models from product descriptions. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 45–54, New York, NY, USA, 2012. ACM.

[Acher *et al.*, 2013a] Mathieu Acher, Benoit Baudry, Patrick Heymans, Anthony Cleve, et Jean-Luc Hainaut. Support for reverse engineering and maintaining feature models. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 20:1–20:8, New York, NY, USA, 2013. ACM.

[Acher *et al.*, 2013b] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, et Philippe Lahire. Extraction and evolution of architectural variability models in plugin-based systems. *Software & Systems Modeling (SoSyM)*, page 27 p., Juillet 2013.

[Al-Msie'deen *et al.*, 2013a] Ra'Fat Al-Msie'deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, et Hamzeh Eyal Salman. Feature location in a collection of software product variants using Formal Concept Analysis. In *ICSR*, pages 302–307, 2013.

[Al-Msie'deen *et al.*, 2013b] 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 *Proceedings of The 25th International Conference on Software Engineering and Knowledge Engineering*, pages 244–249, 2013.

[Ambler, 2004] Scott W. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0.* Cambridge University Press, 2004.

[Apel and Kästner, 2009] Sven Apel et Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, July/August 2009. Refereed Column.

[Apel *et al.*, 2013] Sven Apel, Don Batory, Christian Kästner, et Gunter Saake. A development process for feature-oriented product lines. In *Feature-Oriented Software Product Lines*, pages 17–44. Springer Berlin Heidelberg, 2013.

[Arévalo *et al.*, 2007] Gabriela Arévalo, Anne Berry, Marianne Huchard, Guillaume Perrot, et Alain Sigayret. Performances of galois sub-hierarchy-building algorithms. In *Proceedings of* *the 5th international conference on Formal concept analysis*, ICFCA'07, pages 166–180, Berlin, Heidelberg, 2007. Springer-Verlag.

[Azmeh *et al.*, 2011] Zeina Azmeh, Marianne Huchard, Amedeo Napoli, Mohamed Rouane Hacene, et Petko Valtchev. Querying relational concept lattices. In *CLA*, pages 377–392, 2011.

[Azmeh, 2011] Zeina Azmeh. *A Web service selection framework for an assisted SOA*. PhD thesis, LIRMM - University of Montpellier 2, Montpellier, France, October 2011.

[Baader and Distel, 2008] Franz Baader et Felix Distel. A finite basis for the set of EL-implications holding in a finite model. In Medina et Obiedkov [2008], pages 46–61.

[Bachmann and Bass, 2001] Felix Bachmann et Len Bass. Managing variability in software architectures. *SIGSOFT Softw. Eng. Notes*, 26(3):126–132, Mai 2001.

[Barbut and Monjardet, 1970] M. Barbut et B. Monjardet. *Ordre et classification: algèbre et combinatoire*. Collection Hachette université: Méthodes mathématiques des sciences de l'homme. Hachette, 1970.

[Bass *et al.*, 2003] L. Bass, P. Clements, et R. Kazman. *Software Architecture in Practice*. SEI series in software engineering. Addison-Wesley, 2003.

[Basten and Klint, 2009] H. J. Basten et P. Klint. Software language engineering. chapitre DeFacto: Language-Parametric Fact Extraction from Source Code, pages 265–284. Springer-Verlag, Berlin, Heidelberg, 2009.

[Batory *et al.*, 2003] Don Batory, Jia Liu, et Jacob Neal Sarvela. Refinements and multi-dimensional separation of concerns. *SIGSOFT Softw. Eng. Notes*, 28(5):48–57, Septembre 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. Springer-Verlag.

[Bécan *et al.*, 2013] Guillaume Bécan, Mathieu Acher, Benoit Baudry, et Sana Ben Nasr. Breathing ontological knowledge into feature model management. Rapport Technique RT-0441, INRIA, Octobre 2013.

[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, Septembre 2010.

[Berry and Browne, 1999] M.W. Berry et M. Browne. *Understanding Search Engines: Mathematical Modeling and Text Retrieval*. ITPro collection. Society for Industrial and Applied Mathematics, 1999.

[Berry *et al.*, 2012] Anne Berry, Marianne Huchard, Amedeo Napoli, et Alain Sigayret. Hermes: an efficient algorithm for building galois sub-hierarchies. In *CLA*, pages 21–32, 2012.

[Beuche, 2009] Danilo Beuche. Transforming legacy systems into software product lines. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 321–321, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[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.

[Bissyande *et al.*, 2013] Tegawende F. Bissyande, Ferdian Thung, Shaowei Wang, David Lo, Lingxiao Jiang, et Laurent Reveillere. Empirical evaluation of bug linking. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 89–98, Washington, DC, USA, 2013. IEEE Computer Society.

[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.

[Braganca and Machado, 2007] Alexandre Braganca et Ricardo J. Machado. Automating mappings between use case diagrams and feature models for software product lines. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[Budhkar and Gopal, 2012] Shivani Budhkar et Arpita Gopal. Component-based architecture recovery from object oriented systems using existing dependencies among classes. *International Journal of Computational Intelligence Techniques*, 3(1):56–59, 2012.

[Cellier *et al.*, 2008] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, et Olivier Ridoux. Formal Concept Analysis enhances fault localization in software. In Medina et Obiedkov [2008], pages 273–288.

[Chen *et al.*, 2005] Kun Chen, Wei Zhang, Haiyan Zhao, et Hong Mei. An approach to constructing feature models based on requirements clustering. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, RE '05, pages 31–40, Washington, DC, USA, 2005. IEEE Computer Society.

[Chen *et al.*, 2009] Lianping Chen, Muhammad Ali Babar, et Nour Ali. Variability management in software product lines: a systematic review. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 81–90, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[Chikofsky and Cross II, 1990] Elliot J. Chikofsky et James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, Janvier 1990.

[Classen *et al.*, 2008] Andreas Classen, Patrick Heymans, et Pierre-Yves Schobbens. What's in a feature: a requirements engineering perspective. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[Clements and Northrop, 2002] Paul Clements et Linda Northrop. *Software Product Lines: Practices and Patterns*. The SEI series in software engineering. Addison Wesley Professional, 2002.

[Conejero *et al.*, 2012] José M. Conejero, Eduardo Figueiredo, Alessandro Garcia, Juan Hernández, et Elena Jurado. On the relationship of concern metrics and requirements maintainability. *Inf. Softw. Technol.*, 54(2):212–238, Février 2012.

[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. IEEE Computer Society.

[Cullum and Willoughby, 2002] J.K. Cullum et R.A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations: Volume 1, Theory*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 2002.

[Czarnecki and 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.

[Czarnecki *et al.*, 2006] Krzysztof Czarnecki, Chang Hwan Peter Kim, et Karl Trygve Kalleberg. Feature models are views on ontologies. In *SPLC*, pages 41–51, 2006.

[Davril *et al.*, 2013] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, et Patrick Heymans. Feature model extraction from large collections of informal product descriptions. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 290–300, New York, NY, USA, 2013. ACM.

[Deerwester *et al.*, 1990] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, et Richard A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[Diaz *et al.*, 2013] Diana Diaz, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Silvia Takahashi, et Andrea De Lucia. Using code ownership to improve ir-based traceability link recovery. In *ICPC*, pages 123–132, 2013.

[Dit *et al.*, 2011] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, et Giuliano Antoniol. Can better identifier splitting techniques help feature location? In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 11–20, Washington, DC, USA, 2011. IEEE Computer Society.

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

[Dolques *et al.*, 2012] Xavier Dolques, Marianne Huchard, Clémentine Nebut, et Philippe Reitz. Fixing generalization defects in UML use case diagrams. *Fundam. Inf.*, 115(4):327–356, Décembre 2012.

[Dubinsky *et al.*, 2013] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, et Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *CSMR*, pages 25–34, 2013.

[Dumais *et al.*, 1988] Susan T. Dumais, George W. Furnas, Thomas K. Landauer, Scott Deerwester, et Richard Harshman. Using latent semantic analysis to improve access to textual information. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '88, pages 281–285, New York, NY, USA, 1988. ACM.

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

[Duszynski *et al.*, 2011] Slawomir Duszynski, Jens Knodel, et Martin Becker. Analyzing the source code of multiple software variants for reuse potential. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 303–307, Washington, DC, USA, 2011. IEEE Computer Society.

[Eisenberg and 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. IEEE Computer Society.

[Falleri and Dolques, 2010] Jean-Rémy Falleri et Xavier Dolques. erca - eclipse's relational concept analysis - google project hosting, 2010.

[Falleri *et al.*, 2010] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, et M. Dao. Automatic extraction of a wordnet-like identifier network from software. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 4–13, Washington, DC, USA, 2010. IEEE Computer Society.

[Fellbaum, 1998] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Language, speech, and communication. MIT Press, 1998.

[Ferré *et al.*, 2005] Sébastien Ferré, Olivier Ridoux, et Benjamin Sigonneau. Arbitrary relations in Formal Concept Analysis and logical information systems. In *ICCS*, éditeurs Frithjof Dau, Marie-Laure Mugnier, et Gerd Stumme, volume 3596 de *Lecture Notes in Computer Science*, pages 166–180. Springer, 2005.

[Frakes and Baeza-Yates, 1992] W.B. Frakes et R. Baeza-Yates. *Information retrieval: data structures and algorithms*. Prentice Hall, 1992.

[Ganter and Wille, 1997] Bernhard Ganter et Rudolf Wille. F*ormal* C*oncept* A*nalysis: Mathematical Foundations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st édition, 1997.

[Ganter *et al.*, 2005] éditeurs Bernhard Ganter, Gerd Stumme, et Rudolf Wille. *Formal Concept Analysis, Foundations and Applications*, volume 3626 de *Lecture Notes in Computer Science*. Springer, 2005.

[Godin and Mili, 1993] Robert Godin et Hafedh Mili. Building and maintaining analysis-level class hierarchies using galois lattices. *SIGPLAN Not.*, 28(10):394–410, Octobre 1993.

[Gomaa, 2004] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[Gotel and Finkelstein, 1994] O. Gotel et A. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101, 1994.

[Grechanik *et al.*, 2007] Mark Grechanik, Kathryn S. McKinley, et Dewayne E. Perry. Recovering and using use-case-diagram-to-source-code traceability links. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 95–104, New York, NY, USA, 2007. ACM.

[Griss *et al.*, 1998] M. L. Griss, J. Favaro, et M. d' Alessandro. Integrating feature modeling with the rseb. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 76–, Washington, DC, USA, 1998. IEEE Computer Society.

[Grossman and Frieder, 2004] D.A. Grossman et O. Frieder. *Information Retrieval: Algorithms and Heuristics*. Kluwer international series in engineering and computer science. Springer, 2004.

[Hacene *et al.*, 2013] Mohamed Rouane Hacene, Marianne Huchard, Amedeo Napoli, et Petko Valtchev. Relational concept analysis: mining concept lattices from multi-relational data. *Ann. Math. Artif. Intell.*, 67(1):81–108, 2013.

[Haiduc *et al.*, 2010] Sonia Haiduc, Jairo Aponte, et Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 223–226, New York, NY, USA, 2010. ACM.

[Halmans and Pohl, 2004] Günter Halmans et Klaus Pohl. Communicating the variability of a software-product family to customers. *Inform., Forsch. Entwickl.*, 18(3-4):113–131, 2004.

[Hamdouni *et al.*, 2010] Alae-Eddine El Hamdouni, Abdelhak Seriai, et Marianne Huchard. Component-based architecture recovery from object oriented systems via relational concept analysis. In *CLA*, pages 259–270, 2010.

[Haslinger *et al.*, 2011] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, et Alexander Egyed. Reverse engineering feature models from programs' feature sets. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 308–312, Washington, DC, USA, 2011. IEEE Computer Society.

[Haslinger, 2012] Evelyn Nicole Haslinger. Reverse engineering feature models from program configurations. Master's thesis, Johannes Kepler University Linz, Linz, Austria, September 2012.

[Huchard *et al.*, 2007] M. Huchard, M. Rouane Hacene, C. Roume, et P. Valtchev. Relational concept discovery in structured datasets. *Annals of Mathematics and Artificial Intelligence*, 49(1-4):39–76, Avril 2007.

[Jacobson and Ng, 2004] Ivar Jacobson et Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[Jacobson *et al.*, 1997] Ivar Jacobson, Martin Griss, et Patrik Jonsson. *Software reuse: architecture process and organization for business success.* ACM Press books. ACM Press, 1997.

[Jacobson, 1992] Ivar Jacobson. *Object-oriented software engineering: a use case driven approach.* ACM Press Series. ACM Press, 1992.

[Kang *et al.*, 1998] Kyo Chul Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, et Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Software Eng.*, 5:143–168, 1998.

[Kang, 1990] Kyo-Chul Kang. *Feature-oriented Domain Analysis (FODA): Feasibility Study; Technical Report CMU/SEI-90-TR-21 - ESD-90-TR-222.* Software Engineering Inst., Carnegie Mellon Univ., 1990.

[Kaytoue *et al.*, 2010] Mehdi Kaytoue, Zainab Assaghir, Amedeo Napoli, et Sergei O. Kuznetsov. Embedding tolerance relations in formal concept analysis: an application in information fusion. In *CIKM*, éditeurs Jimmy Huang, Nick Koudas, Gareth J. F. Jones, Xindong Wu, Kevyn Collins-Thompson, et Aijun An, pages 1689–1692. ACM, 2010.

[Kebir *et al.*, 2012] Selim Kebir, Abdelhak-Djamel Seriai, Sylvain Chardigny, et Allaoua Chaoui. Quality-centric approach for software component identification from object-oriented code. In *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, WICSA-ECSA '12, pages 181–190, Washington, DC, USA, 2012. IEEE Computer Society.

[Krueger, 1992] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, Juin 1992.

[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.

[Kuhn, 2009] Adrian Kuhn. Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 175–178, Washington, DC, USA, 2009. IEEE Computer Society.

[Laguna and Crespo, 2013] Miguel A. Laguna et Yania Crespo. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Sci. Comput. Program.*, 78(8):1010–1034, 2013.

[Linsbauer *et al.*, 2013] Lukas Linsbauer, E. Roberto Lopez-Herrejon, et Alexander Egyed. Recovering traceability between features and code in product variants. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 131–140, New York, NY, USA, 2013. ACM.

[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. ACM.

[Loesch and Ploedereder, 2007] Felix Loesch et Erhard Ploedereder. Optimization of variability in software product lines. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 151–162, Washington, DC, USA, 2007. IEEE Computer Society.

[Lopez-Herrejon and Batory, 2001] Roberto E. Lopez-Herrejon et Don S. Batory. A standard problem for evaluating product-line methodologies. In *GCSE*, éditeur Jan Bosch, volume 2186 de *Lecture Notes in Computer Science*, pages 10–24. Springer, 2001.

[Lucia *et al.*, 2012] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, et Sebastiano Panichella. Using IR methods for labeling source code artifacts: Is it worthwhile? In *ICPC*, pages 193–202, 2012.

[Marcus and Maletic, 2003] Andrian Marcus et Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[Marcus *et al.*, 2004] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, et Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[McGregor *et al.*, 2002] John D. McGregor, Linda M. Northrop, Salah Jarrad, et Klaus Pohl. Guest editors' introduction: Initiating software product lines. *IEEE Softw.*, 19(4):24–27, Juillet 2002.

[Medina and Obiedkov, 2008] éditeurs Raoul Medina et Sergei A. Obiedkov. *Formal Concept Analysis, 6th International Conference, ICFCA 2008, Montreal, Canada, February 25-28, 2008, Proceedings*, volume 4933 de *Lecture Notes in Computer Science*. Springer, 2008.

[Metzger *et al.*, 2007] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, et Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *RE*, pages 243–253, 2007.

[Müller *et al.*, 1993] Hausi A. Müller, Scott R. Tilley, et Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 217–226. IBM Press, 1993.

[Paškevičius *et al.*, 2012] Paulius Paškevičius, Robertas Damaševičius, Eimutis karčiauskas, et Romas Marcinkevičius. Automatic extraction of features and generation of feature models from Java programs. *Information Technology and Control*, pages 376 – 384, 2012.

[Pine, 1993] Joseph Pine. *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press, 1993.

[Pohl *et al.*, 2005] Klaus Pohl, Günter Böckle, et Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Poshyvanyk and Marcus, 2007] Denys Poshyvanyk et Andrian Marcus. Combining Formal Concept Analysis with Information Retrieval for concept location in source code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 37–48, Washington, DC, USA, 2007. IEEE Computer Society.

[Poshyvanyk *et al.*, 2006] Denys Poshyvanyk, Andrian Marcus, Václav Rajlich, Yann-Gaël Guéhéneuc, et Giuliano Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *ICPC*, pages 137–148, 2006.

[Prediger and Wille, 1999] Susanne Prediger et Rudolf Wille. The lattice of concept graphs of a relationally scaled context. In *ICCS*, éditeurs William M. Tepfenhart et Walling R. Cyre, volume 1640 de *Lecture Notes in Computer Science*, pages 401–414. Springer, 1999.

[Rajlich and Wilde, 2002] Václav Rajlich et Norman Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 271–, Washington, DC, USA, 2002. IEEE Computer Society.

[Rakic and Budimac, 2013] Gordana Rakic et Zoran Budimac. Introducing enriched concrete syntax trees. *CoRR*, abs/1310.0802, 2013.

[Rubin and 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. ACM.

[Rubin and Chechik, 2013a] Julia Rubin et Marsha Chechik. A framework for managing cloned product variants. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1233–1236, Piscataway, NJ, USA, 2013. IEEE Press.

[Rubin and Chechik, 2013b] Julia Rubin et Marsha Chechik. A survey of feature location techniques. In *Domain Engineering*, éditeurs Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, et Jorn Bettin, pages 29–58. Springer Berlin Heidelberg, 2013.

[Rubin *et al.*, 2012] Julia Rubin, Andrei Kirshin, Goetz Botterweck, et Marsha Chechik. Managing forked product variants. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 156–160, New York, NY, USA, 2012. ACM.

[Rubira *et al.*, 2005] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, et F. Castor Filho. Exception handling in the development of dependable component-based systems. *Softw. Pract. Exper.*, 35(3):195–236, Mars 2005.

[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.

[Salman *et al.*, 2013] Hamzeh Eyal Salman, Abdelhak-Djamel Seriai, et Christophe Dony. Feature-to-code traceability in a collection of software variants: Combining Formal Concept Analysis and Information Retrieval. In *IRI*, pages 209–216. IEEE, 2013.

[Schmid and Verlage, 2002] Klaus Schmid et Martin Verlage. The economic impact of product line adoption and evolution. *IEEE Softw.*, 19(4):50–57, Juillet 2002.

[She *et al.*, 2011] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, et Krzysztof Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, New York, NY, USA, 2011. ACM.

[She, 2008] Steven She. Feature model mining. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 2008.

[Sridhara *et al.*, 2010] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, et K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[Svahnberg *et al.*, 2005] Mikael Svahnberg, Jilles van Gurp, et Jan Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, Juillet 2005.

[Thüm *et al.*, 2014] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, et Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, 79:70–85, Janvier 2014.

[Tilley *et al.*, 2005] Thomas Tilley, Richard Cole, Peter Becker, et Peter W. Eklund. A survey of formal concept analysis support for software engineering activities. In Ganter et al. [2005], pages 250–271.

[Tizzei *et al.*, 2011] Leonardo P. Tizzei, Marcelo Dias, Cecília M. F. Rubira, Alessandro Garcia, et Jaejoon Lee. Components meet aspects: Assessing design stability of a software product line. *Inf. Softw. Technol.*, 53(2):121–136, Février 2011.

[Tizzei *et al.*, 2012] Leonardo P. Tizzei, Cecília M. F. Rubira, et Jaejoon Lee. A feature-oriented solution with aspects for component-based software product line architecting. In *SEAA'12*, pages 1–10. IEEE, 2012.

[Valtchev *et al.*, 2005] Petko Valtchev, Robert Godin, Rokia Missaoui, Marianne Huchard, Amedeo Napoli, David Grosser, Cyril Roume, Amine M. Rouane-Hacene, Jin Zuo, Céline Frambourg, Laszlo Szathmary, Kamal Nehme, et Awa Diop. Galicia lattice builder home page, 2005.

[van der Linden *et al.*, 2007] Frank van der Linden, Klaus Schmid, et Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer, 2007.

[van Deursen and Klint, 2002] Arie van Deursen et Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[Weiss and 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.

[Weston *et al.*, 2009] Nathan Weston, Ruzanna Chitchyan, et Awais Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 211–220, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[Xue *et al.*, 2012] Yinxing Xue, Zhenchang Xing, et Stan Jarzabek. Feature location in a collection of product variants. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 145–154, Washington, DC, USA, 2012. IEEE Computer Society.

[Xue, 2011] Yinxing Xue. Reengineering legacy software products into software product line based on automatic variability analysis. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1114–1117, New York, NY, USA, 2011. ACM.

[Xue, 2013] Yinxing Xue. *Reengineering Legacy Software Products into Software Product Line.* PhD thesis, National university of singapore, jan. 2013.

[Yang *et al.*, 2009] Yiming Yang, Xin Peng, et Wenyun Zhao. Domain feature model recovery from multiple applications using data access semantics and Formal Concept Analysis. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 215–224, Washington, DC, USA, 2009. IEEE Computer Society.

[Ye *et al.*, 2009] Pengfei Ye, Xin Peng, Yinxing Xue, et Stan Jarzabek. A case study of variation mechanism in an industrial product line. In *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering*, ICSR '09, pages 126–136, Berlin, Heidelberg, 2009. Springer-Verlag.

[Yevtushenko *et al.*, 2006] Serhiy Yevtushenko, Julian Tane, Tim B. Kaiser, Sergei Objedkov, Joachim Hereth Correia, et Heiko Reppe. The concept explorer, 2006.

[Yoshimura *et al.*, 2006] Kentaro Yoshimura, Dharmalingam Ganesan, et Dirk Muthig. Assessing merge potential of existing engine control systems into a product line. In *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems*, SEAS '06, pages 61–67, New York, NY, USA, 2006. ACM.

[Zave and Jackson, 1997] Pamela Zave et Michael Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, Janvier 1997.

[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, Avril 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 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 417–422, Washington, DC, USA, 2012. IEEE Computer Society.

Abstract

The idea of Software Product Line (SPL) approach is to manage a family of similar software products in a reuse-based way. Reuse avoids repetitions, which helps reduce development/maintenance effort, shorten time-to-market and improve overall quality of software. To migrate from existing software product variants into SPL, one has to understand how they are similar and how they differ one from another. Companies often develop a set of software variants that share some features and differ in other ones to meet specific requirements. To exploit existing software variants and build a software product line, a feature model must be built as a first step. To do so, it is necessary to extract mandatory and optional features in addition to associate each feature with its name. Then, it is important to organize the mined and documented features into a feature model. In this context, our thesis proposes three contributions. Thus, we propose, in this dissertation as a first contribution a new approach to mine features from the object-oriented source code of a set of software variants based on Formal Concept Analysis, code dependency and Latent Semantic Indexing. The novelty of our approach is that it exploits commonality and variability across software variants, at source code level, to run Information Retrieval methods in an efficient way. The second contribution consists in documenting the mined feature implementations based on Formal Concept Analysis, Latent Semantic Indexing and Relational Concept Analysis. We propose a complementary approach, which aims to document the mined feature implementations by giving names and descriptions, based on the feature implementations and use-case diagrams of software variants. The novelty of our approach is that it exploits commonality and variability across software variants, at feature implementations and use-cases levels, to run Information Retrieval methods in an efficient way. In the third contribution, we propose an automatic approach to organize the mined documented features into a feature model. Features are organized in a tree which highlights mandatory features, optional features and feature groups (and, or, xor groups). The feature model is completed with requirement and mutual exclusion constraints. We rely on Formal Concept Analysis and software configurations to mine a unique and consistent feature model. To validate our approach, we applied it on three case studies: ArgoUML-SPL, Health complaint-SPL, Mobile media software product variants. The results of this evaluation validate the relevance and the performance of our proposal as most of the features and its constraints were correctly identified.

Keywords: Software Product Line Engineering, Software Product Variants, Re-engineering, Feature location, Feature model, Variability, Formal Concept Analysis, Latent Semantic Indexing, Relational Concept Analysis, Feature documentation, Code comprehension, Use-case diagram.

Résumé

Les lignes de produits logicielles constituent une approche permettant de construire et de maintenir une famille de produits logiciels similaires mettant en œuvre des principes de réutilisation. Ces principes favorisent la réduction de l'effort de développement et de maintenance, raccourcissent le temps de mise sur le marché et améliorent la qualité globale du logiciel. La migration de produits logiciels similaires vers une ligne de produits demande de comprendre leurs similitudes et leurs différences qui s'expriment sous forme de caractéristiques (features) offertes. Dans cette thèse, nous nous intéressons au problème de la construction d'une ligne de produits à partir du code source de ses produits et de certains artefacts complémentaires comme les diagrammes de cas d'utilisation, quand ils existent. Nous proposons des contributions sur l'une des étapes principales dans cette construction, qui consiste à extraire et à organiser un modèle de caractéristiques (feature model) dans un mode automatisé. La première contribution consiste à extraire des caractéristiques dans le code source de variantes de logiciels écrits dans le paradigme objet. Trois techniques sont mises en œuvre pour parvenir à cet objectif : l'Analyse Formelle de Concepts, l'Indexation Sémantique Latente et l'analyse des dépendances structurelles dans le code. Elles exploitent les parties communes et variables au niveau du code source. La seconde contribution s'attache à documenter une caractéristique extraite par un nom et une description. Elle exploite le code source mais également les diagrammes de cas d'utilisation, qui contiennent, en plus de l'organisation logique des fonctionnalités externes, des descriptions textuelles de ces mêmes fonctionnalités. En plus des techniques précédentes, elle s'appuie sur l'Analyse Relationnelle de Concepts afin de former des groupes d'entités d'après leurs relations. Dans la troisième contribution, nous proposons une approche visant à organiser les caractéristiques, une fois documentées, dans un modèle de caractéristiques. Ce modèle de caractéristiques est un arbre étiqueté par des opérations et muni d'expressions logiques qui met en valeur les caractéristiques obligatoires, les caractéristiques optionnelles, des groupes de caractéristiques (groupes ET, OU, OU exclusif), et des contraintes complémentaires textuelles sous forme d'implication ou d'exclusion mutuelle. Ce modèle est obtenu par analyse d'une structure obtenue par Analyse Formelle de Concepts appliquée à la description des variantes par les caractéristiques. L'approche est validée sur trois cas d'étude principaux : ArgoUML-SPL, Health complaint-SPL et Mobile media. Ces cas d'études sont déjà des lignes de produits constituées. Nous considérons plusieurs produits issus de ces lignes comme s'ils étaient des variantes de logiciels, nous appliquons notre approche, puis nous évaluons son efficacité par comparaison entre les modèles de caractéristiques extraits automatiquement et les modèles de caractéristiques initiaux (conçus par les développeurs des lignes de produits analysées).

Mots clefs: Ingénierie des lignes de produits, variante de logiciel, Réingénierie, identification de caractéristique, modèle de caractéristiques, Variabilité, Analyse Formelle de Concepts, Indexation Sémantique Latente, Analyse Relationnelle de Concepts, Documentation de caractéristiques, Compréhension du code, Diagramme de cas d'utilisation.