REENGINEERING LEGACY SOFTWARE PRODUCTS INTO SOFTWARE PRODUCT LINE

YINXING XUE

(B.Eng. Wuhan University, China)

(M.Eng. Wuhan University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

Jan 2013

REENGINEERING LEGACY SOFTWARE PRODUCTS INTO SOFTWARE PRODUCT LINE

Approved by:

A/P Stan Jarzabek, Advisor

A/P Jingsong Dong

A/P Siau-Cheng Khoo,

External Referee: Michael W. Godfrey

Date : Jan. 2013

Acknowledgements

During my journey of pursuing a Ph.D., first I would like to thank my supervisor, A/P Stan Jarzabek. He brought me into the domain of software product line and software maintenance. And thanks to his guidance and encouragement, I found the suitable topic and learned the methods to do research. Besides, Prof. Stan also taught me a lot in academic writing, from which I will benefit for all my life.

I would also like to thank Dr Zhenchang Xing, who I worked close with. He taught me in some many aspects: paper writing, presentation skills, or even programming skills. Without his help, I think I would have more lessons in my Ph.D. study. And in the process of implementing our research tools, he even gave me very detailed technical help, which took much time from him.

Thanks to my families' support, I can focus on my research. Especially, I would thank my wife, who encouraged me a lot when my research progress did go well. And my parents also supported and encouraged me a lot. They also helped to take care of my daughter when I was busy with my research work. For my daughter, I also wish my Ph.D thesis is a gift to her and she will be interested in science.

I would also like to thank the thesis committee members, A/P Jingsong Dong, A/P S Siau-Cheng Khoo, for their time in reading and commenting on my thesis. I also appreciate the efforts from the co-authors of the papers I have written as part of this thesis: Prof. Xing Pen (Fudan University, Shanghai), Mr Pengfei Ye (SAP Shanghai), and Prof. Hongyu Zhang (Qsinghua University, Beijing) for their feedback and input.

Finally, it was not about the outcome of obtaining a PhD but instead it was about the process of getting there! Thanks to the help from my supervisors, families and friends, I can say the following sentence to myself:

"All those years he suffered, those were the best years of his life because they made him who he was." ---------- Movie quote from: Little Miss Sunshine (2006)

Table of Contents

Table of Contents

Summary

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. A number of open problems must be solved for SPL to have wide-spread impact on software practice. One of them is to understand and manage variability in software artefacts. To migrate from existing software products into SPL, one has to understand how they are similar and how they differ one from another. In current practice, such analysis is done mostly manually, with some help of clone detection tools. We propose higher level of automation, and a sandwich approach that consolidates feature knowledge from top-down domain analysis with bottom-up analysis of code similarities in subject software products. Our proposed method integrates model differencing, clone detection, and information retrieval techniques, which can provide a systematic means to reengineer the legacy software products into SPL based on automatic variability analysis. Once the variability among the different product variants have been recovered and understood, SPL core assets are built to facilitate reuse. In that area, our contribution is in proposing effective strategies for managing variability in core assets. We analyzed benefits and trade-offs involved in strategies based on applying multiple traditional variability techniques, and in applying a uniform variability technique of XML-based Variant Configuration Language (XVCL). Our proposed strategies have been evaluated in an industrial project and a number of lab case studies.

List of Tables

List of Figures

List of Figures

1 Introduction

The Software Product Line (SPL) approach aims at improving software productivity and quality by relying on much similarity that exists among software systems and relevant development process [33]. The idea of SPL approach is to manage a family of similar products in a reuse-based way. In last two decades, SPL has been an active research area in software engineering [30,152]. 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 similar, but they also differ in customer-specific features. This creates possibility for reuse. Reuse avoids repetitions, which helps reduce development/maintenance effort, shorten time-to-market and improves overall quality of software [70].

In an SPL, core assets [13,127] are identified and built. Product variants are derived from core assets. Variability among variants is described in terms of *features* [81]. Ideally, by configuring required variant features, we would like to be able to derive a custom product from SPL core assets in automated way. Before SPL has the actual impact on software practice, a number of open problems must be solved. Those open problems include how to discover the variability among the product variants, how to model variability and commonality, how to handle the variability and also how to evaluate the architecture of SPL.

1.1 Research Problems

To reengineer an existing family of legacy systems into SPL, several important prerequisites must be satisfied [111]. First, variability among the product variants should be explicitly identified and must be systematically managed. Second, we should be able to derive a new software product from reusable components, so–called SPL core assets. Thus, understanding the commonality and variability in existing software products constitutes the first step towards building core assets for reuse in SPL.

Given an existing family of legacy product variants, the first step in *extractive approac*h [105] to building an SPL is to understand the variability among the products, as they provide a basis for scoping an SPL [111], and then to design first-cut SPL core assets. From our previous industrial case study on variability management [182] and a study of open-source project [75], we found that it was rare that the legacy products have well-documented artifacts describing variability in details. Considering WingSoft Financial Management System (WFMS) [176,182], the documents for the major versions were available, but for those minor versions the information could only be reverse-engineered from source or recalled by the original developers.

In the thesis, we address the following research questions related to re-engineering legacy code into SPL:

RQ1. Given requirements for product variants, how do we identify the common and variant requirements among them?

RQ2. How are the product variants different at the implement level?

RQ3. Once we know the differences in feature and code in product variants, which variant features configure which code variants?

Once the variability among the product variants has been identified, a wide range of variability techniques can be applied to design SPL core assets. The role of variability techniques is to make core assets reusable in multiple product variants. Due to variability in requirements of product variants, more often than not core assets should be adapted for systematic reuse, not developed or maintained individually. Variability should make such adaptive reuse easy. Examples of variability techniques include CPP [86,128,141], Java conditional compilation [197], commenting out feature code, design patterns [57], parameter configuration files, and a build tool Ant [188], parameter configuration files.

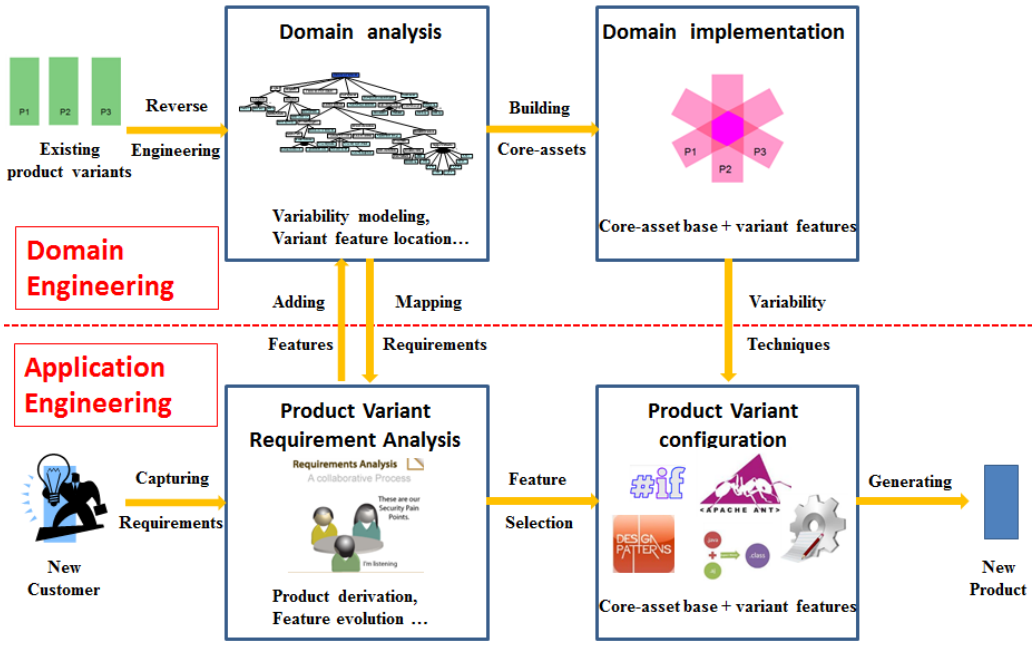

Figure 1.1 An overview of domain engineering and application engineering in extractive approach [94,133]

Figure 1.1 shows two phases of SPL engineering, namely domain engineering and application engineering. Reengineering of existing product variants provides inputs for domain engineering. The feature diagram (variability model) [81] is created during domain analysis, and the core assets are created during domain implementation. During application engineering, developers select variant features for a new product they build and adapt core assets accordingly.

Thus, RQ1 to RQ3 focus on the domain analysis and domain implementation. Here are the extra questions we address in the application engineering to generate products for new customers (after RQ3):

RQ4. What variability techniques are used in industrial SPL?

Our industrial studies revealed that multiple variability techniques are used to tackle different variability situations in SPL. We also analyzed the reasons for the necessity to use multiple variability techniques at the same time. The lessons of adopting multiple variabil-

ity techniques show that in the long run, variability management with multiple variability techniques become difficult to comprehend and maintain for the programmers. To alleviate the problems of multiple variability techniques, we address the final research questions:

RQ5. Can we use a single uniform variability technique instead of multiple techniques? What are the necessary characteristics of such uniform technique and trade-off involved in using it?

1.2 Sketch of the Solution

To answer the RQ1, RQ2 and RQ3, we propose a sandwich approach [177], as shown in Figure 1.2, which consolidates feature knowledge from top-down domain analysis with bottom-up analysis of software clones in subject software product.

To tackle RQ1, we present a model differencing based method to detect changes that occurred to product features in a family of product variants. The primary input to our method is a set of Product Feature Models (PFMs) [180]. A PFM captures all the features and their dependencies in a product variant. We then adapt *GenericDiff* [175], a general framework for model comparison, to compare pair-wisely these PFMs based on both lexical and structural (i.e., dependencies and relationships) similarities of features. We propose a catalog of feature changes that can evolve a PFM, e.g. *rename feature*, *add leaf feature* and so on. Based on the differencing report by *GenericDiff*, we also develop a tool for automatically inferring feature changes according to the catalog we propose.

For RQ2, we use clone detection tool [11] to find the clone candidates that represent the similar variant features. We capture contextual information of clones from Program Dependence Graphs (PDGs) generated by Wala [204]. These PDGs encode data and control dependencies between program statements. We then use graph matching techniques *GenericDiff* to compute a precise characterization of clones in terms of the structural differences and differential properties between their PDGs, from which several patterns of con-

textual differences are recognized [174]. The patterns of contextual differences include *Missing Statement, Missing Branch* and so on [174].

To answer RQ3, we correlate variability recovered from product features models (PFMs) with variability identified from the clones [179]. The underlying intuition of our 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. We propose to incorporate *software differencing*, *Formal Concept Analysis (FCA)*, and *IR* techniques. Software differencing helps to identify distinct features (or code units) in a software product family, which represent corresponding features (or code units) across product variants. FCA then groups distinct features (or code units) into disjoint and minimal partitions by analyzing commonality and differences of product variants. Finally, given a feature partition and the corresponding code-unit partition, Latent Sematic Indexing [41] (LSI) is used to identify code units that implement a specific feature.

For the RQ 4, we first analyze WFMS-PL variant features and present them as a feature diagram [81]. Then, we study variability techniques in WFMS, i.e. Java conditional compilation, commenting out feature code, design patterns [57], parameter configuration files, and a build tool Ant. Finally, we analyze how the granularity and scope of features impact on WFMS components affects the effectiveness of variability techniques [182].

For the RQ5, we conduct lab studies and collect inputs from Fudan Wingsoft Ltd [176], regarding the original WFMS core assets developed by Wingsoft using multiple variability techniques, and core assets in XVCL [71]. We compare the efforts in productivity during domain engineering (i.e., building and evolving core assets), and product derivation. In addition to the above comparative study, we also interview several Wingsoft engineers on the XVCL solution, and summarize their feedbacks and comments in terms of drawbacks and merits of XVCL solution.

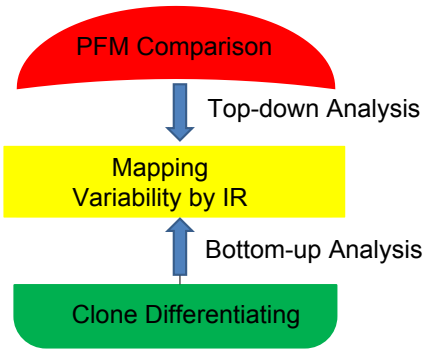

Figure 1.2. The sandwich approach to recovering the variability

1.3 Research Contribution

Existing studies in SPL on domain analysis and requirement engineering to resolve the RQ1 are focused on developing modeling techniques, such as goal model [38] and feature model [83] to capture and analyze requirements, monitoring and tracing requirement changes [66,187] and reasoning about the consistency and configurability of requirement models [22,162]. However, there is a lack of tools that can automatically produce an accurate report of the differences of product variants in terms of feature.

Program differencing methods have long been used for identifying the textual, syntactic and semantic differences between programs [6,69,181], which can also be adopted to address the RQ2 and report the differences between the product variants. However, using program differencing for that purpose would require a pair-wise comparison of any two code fragments of product variants, which is computationally costly. In our work, instead of applying program differencing techniques direct onto the implementation level, we use clone detection for a fast selection of highly similar code fragments that may indicate the variant features, and then use PDG differencing to compute a precise characterization of the differences of those clones.

The RQ3 is essentially a feature location [44,142] or traceability [4,116] problem. Feature location techniques investigate how features are implemented in software artifacts,

such as code, test cases, by using static analysis, e.g. Latent Semantic Indexing (LSI) [113] and concept analysis [48], or dynamic analysis, e.g. execution scenarios [49] and trace intersection [150]. These existing techniques are designed to locate the program elements of a particular feature in a single software system. As in the circumstance of SPL with multiple product variants at hand, it is innovative to take into account these product variants for the feature location problem.

As for variability techniques, previous studies [30,111] have introduced and described these traditional variability techniques, e.g. Java conditional compilation, commenting out feature code, design patterns [57], parameter configuration files, and a build tool Ant. The RQ4 and RQ5 indicate that the industry case study introducing how they can work together is still unavailable.

To sum up, the potential contributions of this dissertation are listed as follows:

1. We propose and implement the tool to automatically compute the difference of requirements of the different product variants. We propose the concept of PFM [180] to model the hierarchy of features contained in products.
2. We combine clone detection techniques and program differencing techniques for the purpose of comparing the similar but different variant features [174]. To better facilitate the comparison of clones based on Program Dependency Graph (PDG), we implement the tool called CloneDifferentiator [173,178].
3. Different from the previous study mainly on feature location in a single product, we focus on locating features by the help of knowledge of the commonality and variability among product variants. We also conduct an empirical study on the product family of Linux [195].
4. We conduct a case study of WFMS, which is a widely-used financial system by major universities in China, to investigate the variability realization techniques

adopted in reality. To some extent, our empirical study reflects the reality of variability management in the small-to-medium software companies.

1.4 Outline

The remainder of thesis is organized as follows: Chapter 2 discusses the related work. Chapter 3 describes the approach that we used to compute the differences of the requirements of the product variants. Chapter 4 presents the approach that we apply to compare the possible variant features – code clones to get their contextual differences. Chapter 5 proposes the method that we adopt to recover the traceability from the variant features in requirements to the difference in code implementing. Chapter 6 summarizes the situation of variability management in real industrial environment. Chapter 7 describes the XVCL solution for variability management and compares it with the one in Chapter 6. Chapter 8 evaluates the whole approach on an artificial product family derived from Berkeley DB. Finally, we conclude and summarize possible future research directions.

2 Preliminaries

This chapter describes the fundamental concepts and techniques on which our work is built on. First, the terms and notations in SPL are introduced and explained. Then the techniques such as clone detection, program differencing and information retrieval techniques, which are used to resolve the research questions, are elaborated in detail.

2.1 Terms and Notations in SPL

As early as in 1970s, some researchers [64,127] proposed the concept "program families" to represent a set of related software products in the same application domain. In the program families, the developers derive the new product by editing from the previous ones, rather than doing it from scratch. But the process of reusing the existing products for the new ones was still done in an ad-hoc way. Until in 1990s, the term "software product line" was officially presented by Software Engineering Institute, Carnegie Mellon University [14]. After that, considering the other business and organizational factors in the process of developing software families for many industrial companies, SEI proposed the term "software product line" as an area referring to software development efforts involved in producing a set of similar but yet different product variants. After that, software product line has become a hot research area [30,152], and many frameworks and development process were presented for the sake of facilitating the ease of development and reducing the cost.

As in automobile industry and many other manufacturing industries "product line" is a refining process to produce an end-product, the SPL also establishes the similar idea by mass customization for software products. Instead of individually developing each product for each customer from scratch, product line engineering develops related variants in a coordinated fashion, developing commonalities between the products only once. Instead of developing a single one-size-fits-all solution that intends to cover all potential customer

needs in a mass market, software product lines provide tail-made solutions for different customers.

2.1.1 Concepts in SPL

Building SPL architecture with the core assets [13,127] also poses extra costs and risks. Usually there are three different approaches to build an SPL: the proactive approach, the reactive approach and the incremental approach. In the proactive one, domain engineers design and develop the core assets before generating the various products. In the reactive one, the core-asset base and variant features are identified and built as the SPL architecture from the existing product variants. For example, our industrial collaborator WingSoft Ltd [182] has many legacy product variants. Building SPL for WingSoft Ltd was adopting the reactive way. Actually, the third incremental approach is mixed by the process of developing the core asset base in stages and the process of develop more product at the same time.

No matter what approach to build an SPL, *feature* is the first class citizen in the feature-oriented software product line to constitute core assets. Usually, there are two kinds of views to represent features: In the view of the internal developers, a feature is often defined as a program function which realizes a group of individual relevant requirements [87]. In the view of external customers, a feature is usually defined as a visible value, quality, or characteristic of software for the end-users [63,81]. In SPL, any product variant can be considered as a set of certain features added to the program base (or the core asset base [133]). The mandatory features refer to the commonly added functions or values shared by all the product variants. Variability existing among variants is described in terms of *variant features* [73,81]. SPL core assets include not only architecture and code components, but also documentation, models, test cases and many other software artifacts, which are relevant to the program base plus variant features and mandatory features.

Chapter 2 Preliminaries

Software product line engineering actually is a two-phase approach composed of *domain engineering* and *application engineering*. Application domain is a software area, which contains the common parts among the similar software systems. For example, those different financial software systems used by the companies are all in the same domain – financial domain. The task of domain engineering is to build the SPL architecture consisting of a core-asset base and the variant features, while the application engineering focus on derivation of the new products by the different customizations of variant features applied onto the core-asset base. These two phases of engineering can have separated life cycles and be maintained by the different engineers, as in Fudan WingSoft Ltd. the core assets are maintained by domain engineers and the new product derivation are conducted by product engineers. In this dissertation, we resolve the key research questions throughout these two phases.

Domain engineering consists of domain analysis, domain design, domain realization and domain testing [111]. Domain analysis aims at recognizing application domains, scoping and bounding them, and identifying commonality and variability among the systems in the domain. Thus, identifying the core-asset base and variant features among product variants is the domain analysis required in building the SPL in a reactive way [129]. However, the domain analysis is actually not limited in the requirements. Instead, the domain analysis should be conducted for all the artifacts in the SPL. For example, in Fudan WingSoft Ltd., the two product variants $WFMS^{Fudan}$ (product variant for Fudan University) and $WFMS^{Shanghai}$ (product variant for Shanghai Jiaotong University) in the WFMS family have two similar features *DelegationLock* and *OperationLock* respectively. Just from the requirement level and design documents, there is no way to distinguish the fact how these two features are different; sometimes they can even be the renamed feature with the same functionality. But by comparing the implementation of these two features, it is possible to

tell how these features are similar and different. In this dissertation, we aim at discovering and validating the core-asset base and variant features not only from the requirements but also from the implementation of product variants.

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. Variability among the product variants in a reactive product line must be identified, modularized or annotated, and evolved throughout the lifecycle of Software Product Line Engineering (SPLE). Such task is called as Variability Management (VM), which is one of the principles fundamental to successful software product line engineering [111]. In software product line engineering, each individual product variant should be not considered and managed by itself. The better way is to look at the product line as a whole ---- the core-asset base and the variation among the individual products. Thus, the domain engineers usually would maintain an all-in-one solution to ease the configuration for any new customers. This all-in-one solution contains all the product variability by adopting the variability techniques.

2.1.2 FODA and Feature Model

Feature-oriented domain analysis (FODA) that our variability analysis is based on was first developed by the Software Engineering Institute in 1990 [81,82]. Originally, the FODA was one possible way towards product line. In the recently twenty years, it actually becomes more and more popular as a defacto prerequisite in constituting product line. In the report, the concept of *feature model* in domain engineering is to represent the so called features within the product family as well as the structural and semantic (require or exclude) relationships between those features [81]. Since then, feature model has even been characterized as "the greatest contribution of domain engineering to software engineering" [36].

Chapter 2 Preliminaries

A *feature model* is a tree-like hierarchy of features. The structural and semantic relationships between a *parent* (or *compound*) feature and its *child* features (or *subfeatures*) can be specified as:

• *And* — if the parent feature is selected, all the subfeatures should be also selected.

• *Alternative* — if the parent feature is selected, only one among the exclusive subfeatures should be selected,

• *Or* — if the parent feature is selected, at least one or at most all subfeatures can be selected,

• *Mandatory* — sometimes called "*Compulsory*", referring to features that required "And"

• *Optional* — features that are optional.

In addition to the above parental relationships between features, there are cross-model constraints allowed. The most common are:

• A *requires* B – The selection of A in a product implies the selection of B.

• A *excludes* B – A and B cannot be part of the same product.

Recently, to enhance the expressiveness, some work [36,37] proposed to make *Or* relationships with **[n:m]** cardinalities, which more specifically denotes that a minimum of **n** features and a maximum of **m** features can be selected.

In addition to the basic or extended cardinality-based feature model, there are many similar models proposed for better modeling the domain knowledge [153]. In this dissertation, we will use and focus on the basic or extended cardinality-based feature model. A feature diagram is a graphical representation of a feature model [81]. As shown in Figure 2.1, we use the FeatureIDE [91], a widely used tool in eclipse, to generate the feature diagram of the on-line Ticket Booking System (TBS). In Figure 2.2, we show the different types of the relationship among these features.

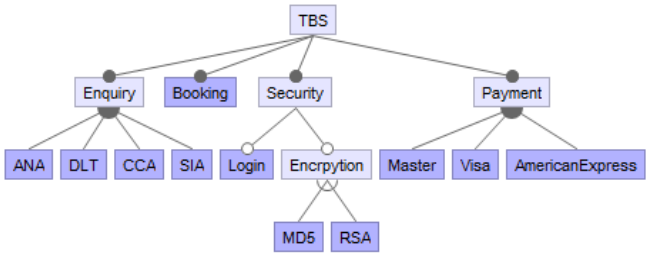

Figure 2.1. The feature diagram of TBS system

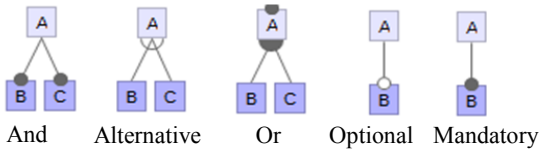

Figure 2.2. The legend for feature diagram of TBS system

```
TBS : Enquiry+ Booking Security Payment+ :: _TBS ;

Enquiry : ANA | DLT | CCA | SIA ;

Security : Login* [Encryption] :: _Security ;

Encryption : MD5 | RSA ;

Payment : Master | Visa | AmericanExpress;
```

Figure 2.3. The grammar for feature diagram of TBS system

The feature *Enquiry* supports the online enquiry for the flight information. The subfeature *ANA* is a function communicating with the interfaces provided by All Nippon Airline. This product family can also dynamically support other features *DTL* (for the Delta Airline), *CCA* (for the China Airline) and *SIA* (for the Singapore Airline) according to the different users' requests. And all the subfeatures are in the *OR* relationship. Any product in the family can support at least one or at most all of the following online payment methods (also *OR* relationship): *Master*, *Visa* and *AmericanExpress*. There are two optional subfeatures *Login* and *Encryption* under the feature *Security*. For the optional feature *Encryption*,

it has two alternative subfeatures *MD5* and *RSA*, one of which should be adopted for data encryption.

For the ease of reasoning and presentation of the feature model, the grammar and propositional formula of feature model are also proposed. A grammar is a compact representation of a propositional formula [17]. As shown in Figure 2.3, model "Payment+" denotes one or more instances of non-terminal "Payment"; "Login*" denotes zero or more. "[Encryption]" denotes optional non-terminal "Encryption". And for this grammar, there is also the corresponding propositional formula, considering the production $r:P_1|\ldots|P_n$, which has n patterns: $P_1 \ldots P_n$.

Pattern	Formula
r	$r \Leftrightarrow \mathbf{choose}_1(P_1,\ldots,P_n)$
r+	$r \Leftrightarrow (P_1 \vee \ldots \vee P_n)$

More mapping and details on the propositional formula of feature model are elaborated in [17]. Figure 2.1 provides the visual representation of feature model, while Figure 2.3 lists the corresponding grammar based on propositional formula. Thus, all these studies on notation and formal specification of feature model facilitate the reasoning on feature model [22,162].

2.2 Clone Detection

Software cloning is an active field of research, which has intrigued the curiosity of researchers for more than 20 years [101,144]. Most software cloning studies focus on the issues such as clone detection, origin of clones, clone classification and clone management. Some studies also focus on the aspect [95] or crosscutting concern mining [25] by using clone detection techniques. Similar to aspects or crosscutting concerns, the scattered features can also be embodied in the duplicated code fragments. In this dissertation, we are

interested in finding those clones relevant to the features, and understanding the commonality and variability among the clone instances inside a clone class.

Most of current available clone detention tools adopt the various techniques: textual comparison, token comparison, metric comparison, comparison of abstract syntax trees (AST), comparison of program dependency graphs (PDG) and other techniques. So far there are plenty of efforts that have been put into the comparison and evaluation of clone detection tools [21,145].

The proper taxonomy or classification of clones will be helpful for the understanding of the reasons, actualities, essence (evil or not) of clones [74,84]. Most of current work on clone classification is based on the following several categories, which are taxonomies based on similarity, taxonomies based on similarity and location, taxonomies based on refactoring opportunities, and taxonomies based on high level structural similarities.

2.2.1 Definition and taxonomy

Software *code clones* are usually the embodiment of the sequences of duplicate code, which recur for multiple times within a program or across different programs. Early in 1998, Baxter et al. [20] defined that a clone is "a program fragment that *is* identical to another fragment". Roy et al. [144] summarized the existing definitions of clone, and argued that these definitions carry some kind of vagueness. In this dissertation, we complete the definition of clones as "two or more clone fragments which satisfy some extent of similarity based on the text, Abstract Syntax Tree (AST), Program Dependency Graph (PDG), metrics, program models or other representations of code".

The most well-known program-text clones can be compared on the basis of the program text that has been copied. We can distinguish the following types of clones accordingly [101] (see Figure 2.4, the central part is the original copy, and the rest parts are the cloned copies.):

• Type 1 is an exact copy without modifications (except for whitespace and comments).

• Type 2 is a syntactically identical copy; only variable, type, or function identifiers have been changed.

• Type 3 is a copy with further modifications; statements have been changed, added, or removed

• Type 4: Functionally, if two blocks of code that conduct the same computation, but implemented through different syntactic variants. For example, a bubble sort algorithm can be written in a *for* () loop or a *do-while* () loop. They have the similar or equivalent behavior, but not in the implementation. We can call these clones semantic clones [99, 108].

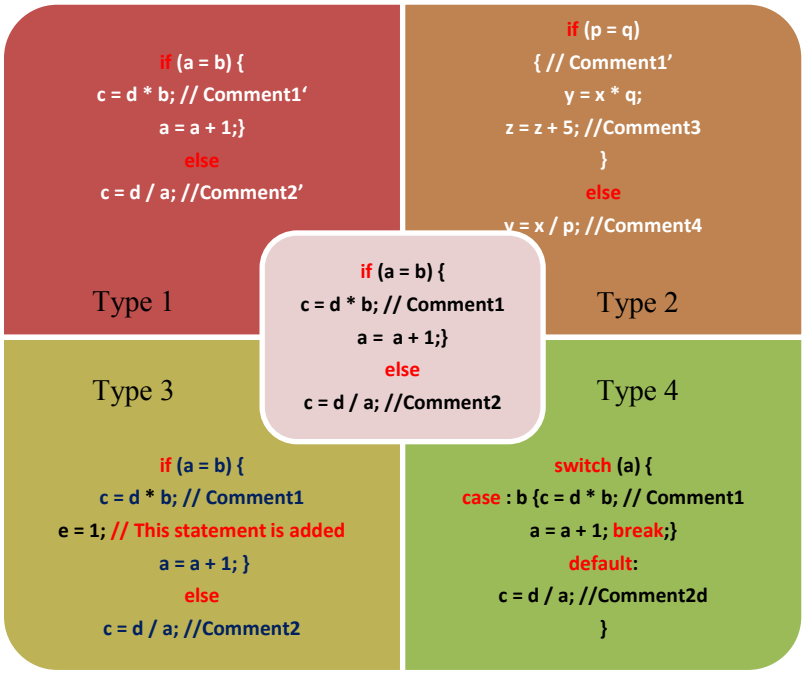

Figure 2.4. The example of Type 1,2,3,4 clones

From the above defined types of clones, we can find some several other kinds of clones which are ramifications of the four standard types of clones. The clone term "**exact**

clones" refers to the Type 1 clone, which just has the modification to comments or whitespace. According to the rule whether the renaming of identifiers is systematic and symmetric, we can divide the Type 2 clones into **parameterized clones** [8] and **renamed clones**, these two kinds. As for the Type 3 clones, depending on whether the activity of renaming identifiers is conducted or not, we can divide them into **near-miss clones** [146] and **gapped clones** [166]. Near-miss clones can include all the Type 2 clones and some Type 3 clones with a slight modification within a statement(s) or even addition and deletion of statement(s). The difference between near-miss clones and gapped clones is that the latter kind just has statement modification: statement insertion or statement deletion.

From the perspective of patterns of recurring clones, **structural clones** [12] mean clones within a syntactic boundary following syntactic structure of a particular language. These boundaries can be function boundary, class boundary, file boundary, directory boundary etc. Structural clones can cover from Type 1 clones to Type 4 clones.

As a special subclass of structural clone, **function clones** [108] refer to the clones that have the whole content of a certain function. According to its similarity level, a function clone can be any type of 1, 2, 3 and 4.

[183] define chained method as a set of methods that hold dependency relations. For given *chained methods*, if each set of the corresponding methods is a code clone, they called the set of *chained methods* **chained clone**. From the definition, it can be concluded that chained clone should be a type of function clones.

The **non-contiguous clone** is an alias for gapped clones [79]. In this kind of code duplication, the different clone instances may just have a few different statements embedded in the common lines of the clones.

In some programs, it is very common that a small block of code recur so frequently that it exists in the multiple source files in an application. In [98], the author gives this kind of

clones a name, **ubiquitous clones**. For example, some flag setting statements or memories disposal statements can be the most usual candidates for this kind of clones.

The term **reordered clones** refer to the clones that have some sequence changes of statements among the different instances as their feature. Given the similarity degree of them, reordered clones have the properties of gapped clones and belong to Type 3 clones, while from the semantic point of view, they have similarity on the semantic level of the codes. So they can be classified as Type 4 clones too.

Another kind of complex clones may be important but not very well known----that is namely **intertwined clones**. An instance of this kind of clone may have the different parts of two code snippets. Put it in another way, it is the two separate different blocks of codes that entangle closely together to form a new single code portion. Discovering these clones is beyond the capability of most current code detection tools as this kind of clones can be a standard subclass of Type 4 clones.

The recent work by Bellon et al. [21] reports a detailed quantitative evaluation of six clone detectors that rely on five different types of program representations. Roy and Cordy [145] present a controlled experiment that evaluates the potential of existing clone detection techniques in handling clones resulting from a set of hypothetical editing scenarios.

Token-based clone detection tool is the fastest, most stable and popular clone detection approach. For example, CCFinder [80] divides the code into tokens and then applies the suffix-tree based sub-string matching algorithm is then used to find the similar subsequences on the transformed token sequence. Tree and graph differencing techniques have been applied for the detection of clones. CloneDR [20] compares abstract syntax tree (AST) of similar code fragments (with same hash index) to determine clones. PDG-based detection tools [67,99,104,112] use subgraph isomorphism to detect similar code fragments. As tree or graph differencing is computationally expensive, these techniques may

not scale to large systems. As a remedy, researchers have investigated reduced representations to approximate program syntax and semantics. Mayrand [120] identifies functions with similar code-metrics values as clones. Gabel et al. [55] encode PDGs in a vector space and then use Locality Sensitive Hashing [58] to cluster similar vectors. CloneMiner [12] exploits frequent item-set mining [61] to detect structural clones across larger program units.

2.2.2 CloneMiner

In this dissertation, we use the clone detection tool CloneAnalyzer [184] to find the simple clones as well as the higher level structural clones. Basit and Jarzabek [11] proposed a new clone type beyond the above introduced four clones from a higher perspective. After detecting clone classes (CC), they move on to the detection of higher level similarity patterns which will present the possible recurring combinations of simple clones.

Following is a list all the cloning abstractions detected by Clone Miner, apart from the simple clone classes (SCS) [184]:

1. repeated groups of simple clones across different methods (simple clone structures or SCS across methods) repeated groups of simple clones across different files (SCS across files)
2. repeated groups of simple clones within a single file (SCS within files)
3. method clone classes (MCC)
4. file clone classes (FCC)
5. repeated groups of method clones across different files (MCS across files)
6. repeated groups of file clones across different directories (FCS across directories)
7. repeated groups of file clones within a single directory (FCS within directories)
8. repeated groups of file clones across different file groups (FCS across groups)
9. repeated groups of file clones within a file group (FCS within groups)

Among the above defined structure clone types, in the sequential chapter, we will use CloneAnalyzer to identify the clones across the different methods, as we concern on the contextual differences of clones in their own methods.

Detecting recurring groups of structural clones from the simple clones is essentially a Frequent Item-set Mining (FIM) problem. Basit applied the same data mining technique used for "market basket analysis". The idea behind this analysis is to find the items that are usually purchased together by different customers from a departmental store. Originally, the input is a list of transactions, each of which consists a list of items bought by the customer for the current transaction. And the output is groups of items which are often bought together. Thus, in our problem domain, analogically the input is a list of simple clone classes (or clone sets), and the output is groups of structural clones in which each such group consists of a list of simple clones appearing together.

The direct application of FIM results in that many mined frequent item-sets are subsets of bigger frequent item-sets. Since our approach mainly considers those biggest frequent item-sets, to remove these subsets the algorithm of "Frequent Closed Itemset Mining" (FCIM) [61] is more suitable. In [12], the algorithm for finding files containing frequent item-sets of simple clone classes is listed. To differentiate the context of the feature relevant code clones, it is helpful to know the information about the repeated groups of simple clones across different methods (SCS across methods). We list the similar algorithm to find methods containing the frequent item-sets in the following Figure 2.5.

Algorithm for finding methods containing the frequent item-set

Procedure FindingFrequentItemSet
Input:

SCC: a list of simple clone classes,

C: the minimum support count, or simply support, of a frequent item-set

S: the minimum size of the item-set

Output:

result: a list of methods containing the recurring groups of simple clone classes.

Body:

```
1.  FIsets = FIM(SCC , C, S)
2.  for each simple clone class sc ∈ SCC do
3.    for each frequent item-set fiset ∈ FIsets do
4.      mset = all methods that contain any instance of sc
5.      for each method m ∈ mset do
6.        if fiset is a subset of the simple clones represented in
              m, keep m in the result
7.        else prune it
8.      end for
9.    end for
10. end for
11. Output the final list result
```

Figure 2.5. Finding methods containing frequent item-sets of SCC

2.3 Program Differencing

For ease of program comprehension, the program differencing techniques are widely used for the analysis of changes made to a system. Maintainers often face the tasks involving analyses of two versions of a program: an old version and a new modified version, e.g. finding the differences for merging two versions in SVN tool. For the context of SPL in this dissertation, the program differencing techniques are also required to compare two product variants generated from the common assets, not only at requirements level but also at implementation level.

In this dissertation, to compare the product variants in requirements and implementation, the program differencing technique is a core part required in our approach.

2.3.1 Status of the art

The program differencing techniques can be further categorized into the following two types: text differencing, model differencing. In earliest decades, the text differencing technique was mainly used to compare programs since at that time a program was just seen as a block of text. The UNIX diff utility [124] is one of the most well-known and popular differencing tools. UNIX diff compares two input text files line by line and outputs differences due to insertion, remove and modification. However, such pure text differencing technique fails to take into account the structure and characteristics of programming language, and overlook behavioral changes corresponding to textual modifications. For example, diff may report changes with no effect on the program behavior, e.g. reordering of class members/methods and changes in comments.

To abstract away the textual noise on the program behavior, the modern mainstream program differencing techniques are mostly based on model differencing algorithms, in which way the program artifacts are usually represented as some forms of models. Yang [181] developed a dynamic programming algorithm to compare the program based on the Abstract Syntax Tree. Apiwattanapong et al. [6] further compared object-oriented programs based on enhanced control flow graphs. Horwitz [69] examines the subgraph isomorphism of intra-method Program Dependency Graphs (PDGs) to detect semantic and textual differences, in terms of unmatched and changed program statements. Since Horwitz's technique was proposed before the invention of Java, it was only defined for a simplified C-like language, not suitable for the object oriented code. But the idea to compare the program based on PDG can be applied to the modern object oriented code. In this dissertation, we adapt GenericDiff framework [175] for the comparison of PDGs of clones to detect the semantic differences of clones at the implementation (object oriented code) level.

Comparing artifacts and detecting their differences is a highly relevant task in software maintenance. Version control systems make use of the differencing algorithm to calculate the delta between a version and its revisions [123]. Xing and Stroulia [171] propose to support evolutionary software development by analyzing the evolution of software design models. Apiwattanpong et al. [6] analyze the impact of structural changes on test cases for the test case selection in regression testing. Kim and Notkin [97] utilize inductive learning to discover and represent systematic code changes. Person et al. [131] support the program equivalence checking and regression test generation based on differential symbolic execution.

In the recent three years, Loh et al. [114] implemented the program differencing tool Logical Structural Diff (LSdiff) that infers systematic structural differences as logic rules. It groups the relevant differences as a single logic rule, and further identifies exceptions that imply missing or inconsistent updates. Maoz et al. [117] implemented the two algorithms for addiff (activity diagram diff) based on binary decision diagrams (BDDs), and integrate it into an Eclipse plug-in. To compute, addiff represents each of the activity diagrams into a module in SMV (the input language in SMV model checker [201]), and applies symbolic model checking for comparison. Duley et al. [45] proposed the Vdiff, a position-independent differencing algorithm for Verilog Hardware Description Language (HDL). Different from the object-oriented code, in Verilog programs the relative ordering between the statements does not matter, and there are also many Boolean expressions to define circuitry. Vdiff extracts ASTs and builds the longest common sequence algorithm to align nodes by the same label. To complement syntactic differencing, it also adopts the modern SAT solver to compare semantic equivalence of two Boolean expressions.

2.3.2 GenericDiff

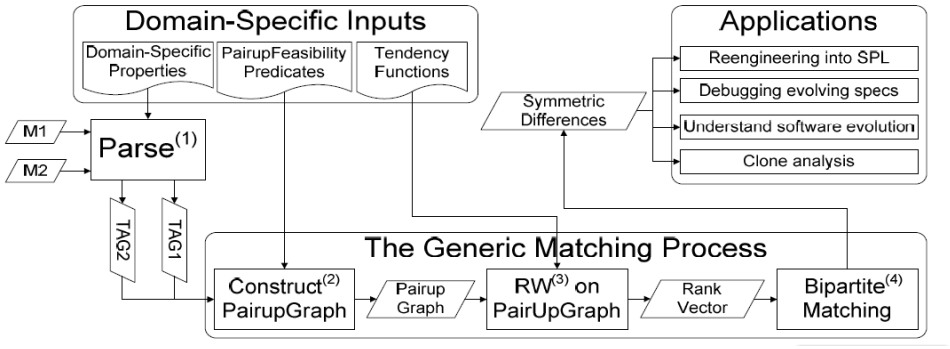

Figure 2.6. The architecture of GenericDiff [175]

In this dissertation, we need to compare the requirements as well as the object oriented code. Instead of using separated differencing algorithm for various software artifacts, we adopt the GenericDiff framework for model comparison [175]. In Figure 2.6, the input of the framework is two models, which can be transformed from source code, requirement specifications or UML diagrams. The output consists of symmetric differences between two compared models. Symmetric differences means if model *a* has one added element *e* than model *b*, the results report two symmetric items: (*null*, *e*, *add*) for model *a* and (*e*, *null*, *delete*) for model *b*. The process of the GenericDiff framework includes four major steps:

1. Parsing the input models into two Typed Attributed Graphs (TAGs). The nodes and edges in TAGs mapped to the entity in the domain models can be defined according to the domain-specific properties.
2. Constructing a PairupGraph$PUG(V_{pu}, E_{pu})$. Suppose we have $TAG_a(V_a, E_a)$ and $TAG_b(V_b, E_b)$ for the two compare models, there exists $V_{pu} \subseteq V_a \times V_b$ and $E_{pu} \subseteq E_a \times E_b$. Pairing up is guided by a set of user-defined feasibility predicate, in which the type compatibility, minimum attribute similarities and the topological constraints are included.

3. Propagating a random walk on the PairupGraph. This step is an iterative process that propagates the distance values from node pair to node pair based on graph structure. The tendency functions are also provided to define constant functions, or functions of the distance values between PairupGraph nodes/edges.
4. Matching bipartite graphs from a rank vector of node pairs outputted by step 3. Each of node pairs inside this rank vector is assigned a numerical correspondence measure representing the matching quality. The stable-marriage algorithm (Gale-Shapley algorithm) is applied to select the optimal matching pairs of nodes/edges.

The interested readers can refer to [175] for the details in each step. Note that the reported symmetric differences are some preliminary results, which entail the further interpretation with the domain knowledge. In Chapter 3 and chapter 4, how the symmetric differences are interpreted is elaborated.

2.3.3 Clone detection vs. program differencing

Program differencing techniques compare two programs at a time, already assuming that there is much similarity among them. However, these techniques are not suitable for clone detection in large systems. Using program differencing for that purpose would require a pair-wise comparison of any two code fragments which results in combinatorial explosion of such operation. AST- or PDG-matching based clone detection techniques [20,99] have all used some hash functions to prune the space of pair-wise differencing. Even then, they may still not scale well to large systems. In this dissertation, instead of applying program differencing techniques within clone detection process, we use clone detection for a fast selection of highly similar code fragments and then use PDG differencing to compute a precise characterization of the differences of those clones.

2.4 Information Retrieval for Feature Location

The information retrieval techniques are widely used in the feature location area to identify the feature/concept-relevant resource [44,142]. In this dissertation, we are concerned about the variant features (variability) and the corresponding code among the product features. We aim at using the commonality and variability analysis to facilitate feature location job in the context of product family or product line.

The software artifacts, such as requirement documents, source code, can be indexed into a document space according to the occurrences of terms within these artifacts. The query can be also indexed and projected onto this document space. By using similarity measure such as cosine similarity, the documents inside document space are compared with this indexed query to get a ranking list for matching.

2.4.1 Vector Space Model

In the Vector Space Model (VSM), documents and queries are represented as vectors of terms that occur within documents in a collection [65]. VSM begins with a term-document matrix, A, 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, a_{ij}, denoting a measure of the weight of the ith term in the jth document. As a document often only contains a small subset of the total terms in the document collection, this term-document matrix is usually very large and very sparse. The weight a_{ij} is actually defined according to the value of term frequency tf_{ij} for the ith term in the jth document. Specifically, we can write:

$$a_{ij} = l(i,j) \cdot g(i)$$

where a local term weight, $l(i,j)$, denoting the relative frequency of the ith term in the jth document, and a global weight, $g(i)$, denoting the relative frequency of the ith term within the entire collection of documents.

Some common local weighting functions [23] are listed as follow:

- Binary: $l(i,j) = 1$ if $tf_{ij} > 0$ or else 0
- TermFrequency: $l(i,j) = tf_{ij}$
- Log: $l(i,j) = \log(tf_{ij} + 1)$

Some common global weighting functions [23] are listed as follow:

- Binary: $g(i) = 1$
- Normal: $g(i) = \frac{1}{\sqrt{\sum_j tf_{ij}^2}}$
- Idf: $g(i) = \log_2 \frac{n}{1+df_i}$, df_i is the number of documents in which the *i*th term occurs.

In this dissertation, we use the *TermFrequency* as the local term weight and the *Normal* as the global weighting functions. From a geometric point of view, each column of the term-by-document matrix (the term vector owned by each document) denotes a point in the *m*-space of the terms. And totally this matrix has *n* points in this *m*-space of the terms. Thus, the similarity between two documents (two vectors) can be measured by the cosine of the angle between the two corresponding points in the *m*-space of the terms. Generally, the closer these two points are in the same (general) direction, the more similar these two documents are.

2.4.2 Singular Value Decomposition

In reality, VSM suffers from the two following drawbacks: 1. VSM ignores the possible semantic relationship among the terms; 2. VSM fails to do rank-reduced simplification. In the dictionary, there may be 50000 words. But actually people may only use a necessary subset like 5000 words as there are synonymity and polysemy among all the words. The dictionary case also holds for the collection of documents. Latent Semantic Indexing (LSI)

became an improvement over the simplistic point of view of term matching, taking account into term dependencies [41].

The basic idea of LSI is to exploit the underlying "latent structure" in word usage pattern by using statistical analysis on the co-occurrence of terms. Then the terms with the same or similar usage pattern may be synonyms or near- synonyms, which can be reduced to one *concept*. How to separate such a set of uncorrelated indexing factors or concepts is done by Singular Value Decomposition (SVD) [34].

The LSI has the same steps with VSM for the construction of the term-document matrix *A*. The LSI applies SVD to decompose the matrix *A* 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, which satisfy the following relations (r is the rank of matrix *A*):.

$$A = T \cdot S \cdot D$$

$$T^T \cdot T = D \cdot D^T = I_r$$

$$S_{1,1} \geq S_{2,2} \geq \cdots \geq S_{r,r} > 0, S_{i,j} = 0 \text{ where } i \neq j$$

$$r \leq \min(m, n)$$

LSI allows the optimal approximation for the standard SVD by reducing the rank or truncating the singular value matrix S to size $k \ll r$ [41].

$$A \approx A_k = T_k \cdot S_k \cdot D_k$$

By the above two steps of transformations, i.e. decomposition and approximation, the initially correlated terms are first grouped into a set of concepts at size of r, and then further compressed into an even smaller set of concepts at size of k. The benefits of such process is to capture most of the important underlying "latent structure" in the association of terms and documents, meanwhile to abstract way the noise due to the users' different word usages that are sensitive to the word-based retrieval methods.

The challenges of using LSI are twofold: the costly computation of SVD and difficulty in determining the optimal value of k.

The computation time for a standard SVD of an $m \times n$ matrix A is as follows:

- Computation of T, S and D: $4m^2n + 8mn^2 + 9n^3$
- Computation of S and D: $4mn^2 + 8n^3$

The above time complexity is usually for the case $m \gg n$. The more general time complexity form should be $O(\min\{m^2n, mn^2\})$. As we use k to approximate the rank r, the time complexity will further be reduced to $O(\min\{k^2n, k^2\mathrm{m}\})$ accordingly.

Thus, the choice of value of k is critical to the performance as well as accuracy of LSI. With much larger repositories (as much as 80,000 terms and 220,000 documents), Dumais et al. [46] reported that better results can be achieved when the concepts (the value of k) is between 235 and 250. But the above observation is especially reasonable for the retrieval on the natural language. As the collection of source code files may have a much larger vocabulary, Poshyvanyk et al. [136] reported that when the value of k is 750, the overall relevant factor for their 9 queries is better compared with k = 300 and 500. But after 750, the result is not stable: some queries got better results and some got worse. It will be desired to see more solid empirical study on choice of the size of concepts.

In Chapter 5, we further explore the relationship between the size of the concepts and, the performance and the accuracy. Our empirical study sheds light on the optimal choice of the size of the concepts for the documents of source code.

3 Understanding Variability in Product Requirements

This chapter is organized based on [180] to answer the RQ1 introduced in Section 1.1. We reviewed the related approaches to understand variant requirements, and proposed our model-differencing based approach to understand the commonality and variability between product variants at the requirement level.

3.1 Introduction

Product variants often evolve from an initial product developed for and successfully used by the first customer, for example WFMS [182] and PAT [157,158]. Figure 3.1 presents five product variants of the Wingsoft Financial Management System[1] (WFMS) [182], spawned from $WFMS^{Fudan}$ developed for Fudan University. The successful deployment of the initial product has attracted new customers, such as Zhejiang University and Shanghai University. WFMSes have now been used in over 100 universities in China.

Initially, WFMS developers copy-pasted and modified existing product variants when building a new product variant. For example, $WFMS^{Chongqing}$ was built by adapting $WFMS^{Fudan}$ and $WFMS^{Zhejiang}$. Such ad hoc reuse becomes problematic as the number of features and the number of product variants grows [126]. Not only do we have to maintain each product variant separately from others, but it also becomes difficult to find and adapt features for reuse in new products [182].

As these problems accumulate, it is worth reengineering product variants into a Software Product Line (SPL) for systematic reuse [33]. Extractive reengineering [105] into SPL is a low cost approach in which the initial reusable core assets include only features

[1] WFMS for Shanghai Jiaotong University: http://www.jdcw.sjtu.edu.cn/wingsoft/index.jsp

already implemented in existing product variants. The first step in extractive approach is therefore to understand common and variant features in existing product variants.

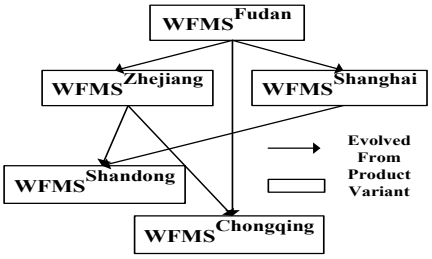

Figure 3.1. The variants of WFMS product family

At first glance, it seems to be an easy task to identify common and variant features in product variants. However, the problem is non-trivial for a big product family with many features that has been evolved for long time. During evolution, feature names and descriptions as well as the dependencies and relationships between features might have been changed. New features might have been added and existing features might have been deleted. The features might have also been split or merged.

The existing work on domain analysis and requirement engineering has been focused on developing modeling techniques, such as goal model [38] and feature model [81] to capture and analyze requirements, monitoring and tracing requirement changes [66,187], and reasoning about the consistency and configurability of requirement models [22,162]. However, there is a lack of automatic tools that can produce an accurate report of feature evolution in a family of product variants.

To automatically identify how these product variants are different in features (see RQ1 in Section 1.1), we present a model differencing based method to detect changes that occurred to product features in a family of product variants. The primary input to our method is a set of Product Feature Models (PFMs). A PFM captures all the features and their dependencies in a product variant (see Figure 3.3). The PFMs can be provided by system ex-

perts of the subject product variants. They may also be reverse-engineered from the implementations of product variants using feature location methods [3,48,100,134].

We then adapt *GenericDiff* [175], a general framework for model comparison, to compare pair-wisely these PFMs based on both lexical and structural (i.e., dependencies and relationships) similarities of features. We propose a catalog of feature changes that can evolve a PFM, namely *rename feature*, *add leaf feature*, *remove leaf feature*, *move feature*, *split feature* and *merge features*. Based on the differencing report by *GenericDiff*, we also developed a tool for automatically inferring feature changes according to the catalog we propose.

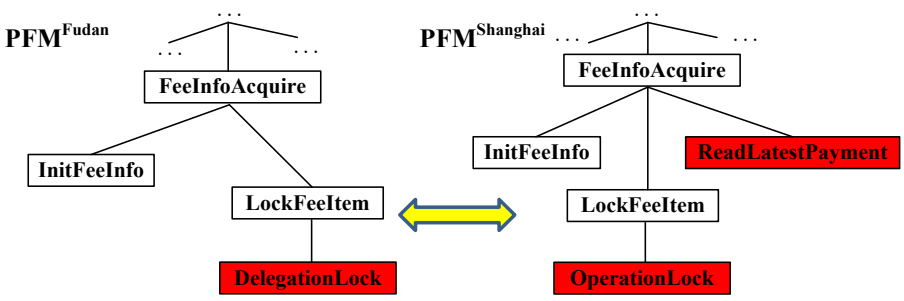

Figure 3.2. Comparison of two PFMs

Figure 3.2 illustrates the partial PFMs (PFM^{Fudan} and $PFM^{Shanghai}$) of two product variants $WFMS^{Fudan}$ and $WFMS^{Shanghai}$ in the WFMS family. Our approach reports that the feature **ReadLatestPayment** is only present in $PFM^{Shanghai}$. Furthermore, it reports that the feature **DelegationLock** in PFM^{Fudan} and the feature **OperationLock** in $PFM^{Shanghai}$ can be a) the same feature but with different names in two product variants or b) they can represent different features.

The novelty and contributions of this work are listed as follows:

1. We bridge the gap caused by a lack of automatic tools that can produce an accurate report of feature evolution in a family of product variants.

2. The existing domain analysis focuses mainly on change tracing and reasoning on the requirements, we propose a model-differencing based approach.
3. The evaluation is conducted on real product line WFMS, and a controlled experiment is also designed to evaluate the scalability of our method with a large volume of synthesized PFMs. And the preliminary results are promising.

3.2 Related Work

Goal model [38] is often used in requirement engineering to capture functional (hard) and non-functional (soft) requirements and their dependencies. Hassine et al. [66] applies slicing and dependency analysis to Use Case Map [26] to identify the impact of requirement changes on the system. Zowghi et al. [187] presents a logical framework for modeling and reasoning about the consistency and completeness of the requirements. Lormans [115] developed a methodology to monitor the evolution of requirements and reconstruct requirement traceability.

Feature model [81] is commonly used to represent common and variant requirements in SPL. Thüm et al. [162] utilize SAT solver to classify the evolution of feature model based on how the configurability of the model has changed. Dhungana et al. [43] proposed a model-driven approach to product line evolution. Their approach supports merging of model fragments into a complete variability model as well as the consistency checking and co-evolution of models and architecture.

The Product Feature Model (PFM) in this dissertation is similar to the hard-goal model, as a PFM captures the requirements of a particular product variant. However, our research goal is to support reengineering product variants into SPL. At requirement level, we would like to derive a domain feature model [83], representing configurable requirements in an SPL. In this sense, we refer to our input model as Product Feature Model.

Existing work on requirement and domain engineering has been focused on the modeling, management of, and reasoning about requirements and their evolution. In this work, we present a model-differencing based method to detect and analyze feature changes at requirement level in a family of product variants. This work enables the variability analysis and consolidation of product variants in the task of extractive reengineering into SPL.

Researchers have also investigated the comparison and merging of other models. Xing and Stroulia developed *UMLDiff* [172] for comparing UML class models for supporting evolutionary development of object-oriented software design. Godfrey and Zou [60] use origin analysis to detect the merging and splitting of source-code entities at the file-structure level. Demeyer et al. [42] propose a set of heuristics for detecting evolutions from refactorings by applying lightweight, object-oriented metrics to successive versions of a software system.

Nejati et al. [125] present an approach to matching and merging state chart specifications. Treude et al [163] use a high-dimensional search tree to efficiently compare models that can be represented as direct, typed graphs. GenericDiff used in this work is a general framework for comparing various types of models. In addition to product feature model, we have also applied *GenericDiff* to compare the implementation (e.g., Program Dependence Graph) of product variants [174].

Bruntink et al. [25] use clone detection to identify crosscutting concerns, which often implement distinct features. Feature location methods support the recovery of features from product implementations using information retrieval techniques [3,135] or scenario-based dynamic analysis [48,100]. These approaches can be exploited to acquire product feature models from the implementations of product variants.

Alves et al. [2] defines a catalog of feature model refactorings that can be enacted in the extractive reengineering into SPL. They assume that it is known where to apply these re-

factorings in a software product family. The output of this work can be exploited to identify the opportunities in a family of product variants where such refactorings are applicable.

3.3 Comparing PFMs

In this part, we first define the Product Feature Model (PFM) (Section 3.3.1). We then summarize a catalog of feature changes (Section 3.3.2). We discuss how these evolutionary feature changes affect the PFMs lexically and structurally, by which the evolution history of a PFM can be expressed in a set of subsequent changes. Next, we present *GenericDiff* framework and describe how we configure *GenericDiff* to compare PFMs (Section 3.3.3). Finally, we analyze the differencing report by *GenericDiff* to infer the changes of product features as product variant evolves (Section 3.3.4).

3.3.1 The meta-model of product feature model

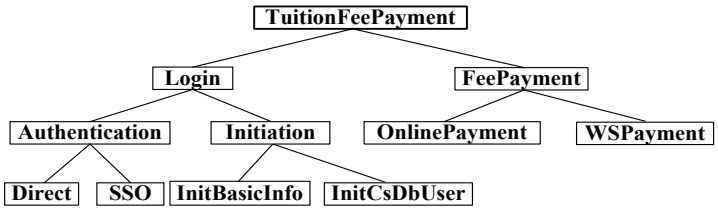

Figure 3.3. A Partial PFM of WFMS^{Shandong}

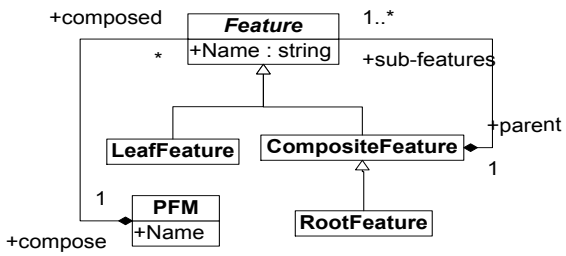

Figure 3.4. The meta-model of PFM

Figure 3.3 shows a partial PFM of the product variant WFMS^{Shandong} in the WFMS product family [182]. The rectangle nodes denote features. This PFM has a root feature *TuitionFeePayment* that refers to the whole system. The root feature is decomposed into two

composite features, *Login* and *FeePayment*. The *Login* is decomposed into two composite features, *Authentication* and *Initiation*, which are further decomposed into leaf features that represent different authentication modes and initiation operations, respectively. The *FeePayment* is decomposed into two leaf features that represent two different payment methods, *OnlinePayment* and *WSPayment.* Such PFMs are the inputs to our approach.

Figure 3.4 presents the meta-model of product feature model. The meta-model defines rules that must be followed to build correct PFMs. A PFM forms a hierarchy of product features. Each *Feature* in a PFM must be uniquely identifiable by its *name* property. Note that the feature name can be any free-form text that describes the feature. A PFM must have a *RootFeature*, which is a special *CompositeFeature* that represents the corresponding product variant. A *Feature* can be decomposed into sub-features. A feature that has no sub-features is a *LeafFeature*; otherwise it is a *CompositeFeature*. The root feature has no *parent* feature, while a non-root feature must have a *parent* feature, i.e. belongs to a composite feature.

3.3.2 A catalog of feature changes

We propose a catalog of feature changes that can evolve a PFM. We define four types of atomic changes, namely *rename feature*, *add leaf feature*, *remove leaf feature*, and *move feature*. Furthermore, we define four types of composite changes that can be composed of a sequence of atomic changes: *add feature subtree*, *remove feature subtree*, *split feature*, and *merge feature*. Note that this catalog of feature changes is sufficient to describe the evolution of PFMs. However, it is possible to define other types of feature changes. Our approach can be easily extended to handle the new types of feature changes.

RenameFeature. A consistent naming scheme improves product maintainability, especially when feature names allude to the functions of the product. A feature may be renamed to reflect the underlying implementation changes, the adoption of different technol-

ogies, or the changes of application context. Renaming a feature *f* changes the *name* property of the feature *f* in PFM.

AddFeature. A product can be extended with new features. A new leaf feature *nlf* can be added to an existing feature *f*. This creates a new *parent-subfeature* relation from *f* to *nlf*. If *f* is a leaf feature, it becomes a composite feature after the addition. Adding a composite feature and its descendants, i.e., *AddFeatureSubtree* can be achieved by traversing the subtree and adding leaf features in preorder.

RemoveFeature. A product variant may not have some features. An existing leaf feature can be removed from a PFM. This removes the *parent-subfeature* relation between the removed leaf feature and its *parent* composite feature. A composite feature may become a leaf feature after the removal. Removing a composite feature and its descendants, i.e., *RemovingFeatureSubtree* can be achieved by traversing the subtree and removing leaf features in postorder.

MoveFeature. The feature hierarchy may be reorganized. Moving a feature *f* (leaf or composite) from a source composite feature *sf* to a target feature *tf* changes the *parent-subfeature* relation between *sf* (*tf*) and *f*. But moving does not affect the *parent-subfeature* relations between *f* and its sub-features. Moving a feature can be achieved by removing the feature from a PFM and then adding it somewhere else in the PFM. However, we consider *move* as an atomic change, since it better conveys the intention of the change (i.e., the reorganization of feature hierarchy) than the separate addition and removal.

SplitFeature. A feature *f* can be split into two or more sibling features (including *f*). If *f* is a composite feature, some of its sub-features will be distributed (i.e., moved) to its new sibling features. Splitting feature can be achieved by first adding new sibling features as leaf features and then moving some of the sub-features of *f* to the relevant new sibling features.

MergeFeature. Two or more sibling features (including *f*) can be merged into a single feature *f*, as in opposite to splitting a feature. The sub-features of the merged sibling features will become the sub-features of this single feature after the merging. Merging feature can be achieved by firstly moving all the sub-features of *f*'s sibling features to *f* and then removing these sibling features.

3.3.3 The differencing of product feature models

Given the PFMs $\{PFM_1..PFM_N\}$ of *N* product variants, we apply *GenericDiff* to compute pair-wisely the differences between the product feature models PFM_i and PFM_j $(i \neq j; 1 \leq i, j \leq N)$ of two product variants. In this chapter, we give an overview of *GenericDiff* framework. We explain how we configure *GenericDiff* to compare PFMs. The interested reader is referred to [175] for further information about *GenericDiff* framework.

Figure 2.6 shows the architecture of *GenericDiff*. *GenericDiff* takes as input two models to be compared (in this work, i.e., Product Feature Models) and the specifications of domain-specific properties, pairup feasibility predicates, and random walk tendency functions (three concepts to be made clear below) for the comparison of input models. It casts the problem of model comparison as a problem of recognizing the *Maximum Common Subgraph* (*MCS*) of the two Typed Attributed Graphs (TAGs).

Given two PFMs, PFM_1 and PFM_2, *GenericDiff* parses(1) the input models into typed attributed graphs, TAG_1 and TAG_2, according to the meta-model of PFMs (See Section 3.3.1). The TAG of a PFM forms a *containment tree*, such as the PFM shown in Figure 3.3. The TAG of a PFM consists of three types of graph nodes, corresponding to *RootFeature*, *LeafFeature* and *CompositeFeature* respectively. Graph edges represent *parent-subfeature* relations between features.

A property of model elements and relations declares a characteristic of their instances. Given a meta-model, one needs to select a set of *domain-specific properties* for each ele-

ment and relation type that characterize its instances. During the parsing process, *GenericDiff* collects data from the selected properties of each model element and relation and represents them in a characteristic composite vector attribute associated with the corresponding graph node and edge. This composite vector attribute is a compact representation of the properties of model elements and relations for efficient graph indexing and matching.

A feature in PFM has three properties, namely, *name*, a *parent* feature, and a (possibly empty) set of *sub-features*. In the *containment tree* representation of PFM, we define, for a feature node *f*, a composite vector of two atomic vectors. One atomic vector represents the set of words in the *name* property of *f*. We choose the Jaccard coefficient, an efficient and commonly used metric to measure the similarity between two sets of words S_1 and S_2, i.e., $S_1 \cap S_2 / S_1 \cup S_2$. The other atomic vector is a numeric vector that stores the number of the *sub-features* of *f*. Given two such numeric vectors, $[v]$ and $[v']$, we choose Manhattan (Taxicab) distance, i.e., $|v - v'|$, to measure their similarity. Manhattan distance did not take into account the direction of the path. The usage of Manhattan distance leads to a larger difference than the usage of Euclidean distance under the same case. The edges of the containment tree of PFM have no characteristic vectors, since they simply represent the *parent-subfeature* relations between features.

Given two TAGs, corresponding to the two compared PFMs, *GenericDiff* constructs[(2)] a PairupGraph, i.e., a product of the two compared model graphs. The PairupGraph encodes the graph structure of two compared models. A node (edge) of PairupGraph represents a pair of nodes (edges) of two compared model graphs. The construction of PairupGraph is guided by a set of user-specified *pairup feasibility predicates* to prune the search space according to domain-specific knowledge. For the comparison of PFMs, we simply define the type compatibility of graph nodes, i.e., [*Feature*, *Feature*]. Note that we define the type

compatibility in terms of the super-type *Feature*. Therefore, the mappings between graph nodes of different subtypes, such as *LeafFeature* and *CompositeFeature*, are allowed.

The initial distance value of a pair of graph nodes (edges) is calculated as the Euclidean length of the normalized distance vector of the two graph nodes (edges). *GenericDiff* performs a random walk(3) on the PairupGraph, which is an iterative process that propagates the distance values from node pair to node pair based on graph structure. A random walk on a graph can be described by a probabilistic model that is defined by a set of *random walk tendency functions*. For the comparison of PFMs, we use the default random walk settings provided by *GenericDiff* framework that define the random walk tendency functions as linear functions of the distance values of the relevant node and edge pairs.

The random walk on the PairupGraph outputs a rank vector of graph node pairs, each of which is assigned a numerical correspondence measure, i.e., the measure of the quality of the match it represents. *GenericDiff* constructs a bipartite graph from this rank vector of node pairs and selects an optimal matching(4) using Gale-Shapley algorithm. Finally, given a pair of matched graph nodes, *GenericDiff* builds a bipartite graph of their edges and uses Gale-Shapley algorithm [56] again to map their edges.

3.3.4 Inferring changes to product features

GenericDiff reports a symmetric difference between two compared models. For the comparison of two PFMs, PFM_1 and PFM_2, *GenericDiff* outputs a set M of corresponding (i.e., matched) features that exist in both PFMs, a set UM_1 of features that are unique in PFM_1, and a set UM_2 of features that are unique in PFM_2. Based on the differencing report by *GenericDiff*, we developed a tool for automatically inferring feature changes (as defined in Section 3.3.1) that can evolve PFM_1 into PFM_2 based on the effects of feature changes on the PFMs.

Let *f* be a feature, we define *parent(f)* returns the *parent* feature of *f* and *name(f)* returns the *name* property of *f*. Let $f_1 \in PFM_1$ and $f_2 \in PFM_2$ be a pair of matched features reported by *GenericDiff*, i.e., $(f_1, f_2) \in M$, our tool reports an instance of a particular type of feature change (by tagging (f_1, f_2) with the corresponding change type) as follows:

- if $name(f_1) \neq name(f_2) \Rightarrow < f_1, f_2, Rename >$
- Let $pf_1 = parent(f_1)$ and $pf_2 = parent(f_2)$, if $(pf_1, pf_2) \notin M \Rightarrow < f_1, f_2, Move >$
- Let $sf \in UM_2$, if $parent(f_2) = parent(sf)$ and $\exists (cf_1, cf_2) \in M$, $f_1 = parent(cf_1)$ and $sf = parent(cf_2)$ and $< cf_1, cf_2, Move > \Rightarrow < f_1, f_2, Split >, < f_1, sf, Split >$
- Let $mf \in UM_1$, if $parent(f_1) = parent(mf)$ and $\exists (cf_1, cf_2) \in M$, $mf = parent(cf_1)$ and $f_2 = parent(cf_2)$ and $< cf_1, cf_2, Move > \Rightarrow < f_1, f_2, Merge >, < mf, f_2, Merge >$

To detect feature renaming, our tool simply examines the *name* properties of a pair of matched features. To detect feature move, our tool examines whether the parent features of a pair of matched features are matched. To detect feature splitting and merging, our tool essentially examines if some sub-features of a pair of matched features are moved to some unmatched sibling features of this pair of matched features. Note that our tool does not report the splitting/merging of leaf features, since there is no distinction between the effect of splitting a leaf feature and that of adding some new leaf features.

All the pairs of matched features that have not been tagged with the above four types of changes will be tagged with *unchanged*. Finally, all the unmatched features in UM_1 and UM_2 (excluding those tagged with *Split* and *Merge*) are reported as features to be *Removed* and *Added* respectively. If a composite feature and all its descendants are tagged with *Add* (*Remove*), our tool reports an *AddFeatureSubtree* (*RemoveFeatureSubtree*).

3.4 Evaluation

In this part, we present the empirical evaluation of the proposed approach. More specifically, we investigate two research questions: 1) How accurate is the proposed approach in detecting changes to product features during evolution? 2) How robust is the proposed approach when the PFMs undergo substantial amount of changes during the evolution process?

3.4.1 WFMS case study

We have applied our approach to analyzing the feature evolution in the product family of WingSoft Financial Management Systems (WFMS). The first product of WFMS was developed in 2003 for Fudan University and it has evolved into a product family with more than 100 customers today. This product family includes 26 product variants. All the product variants share 13 common features, such as *Settlement*, *FileLog*, but also differ in other features specific to a given customer, such as *InitCsDbUser* in $WFMS^{Shandong}$ and *SelectByYear* in $WFMS^{Shanghai}$.

Among all the WFMS product variants, $WFMS^{Chongqing}$, $WFMS^{Shandong}$, $WFMS^{Shanghai}$ and $WFMS^{Zhejiang}$ are four major variants. The other product variants have been derived from them with minor changes. Figure 3.1 shows the evolutionary dependencies among the first product $WFMS^{Fudan}$ and these four major variants. Each of the four major product variants has on average 30 features and 50KLOC of Java code. The system expert of WFMS product family provided us the PFMs for the four major product variants, which were then pair-wisely compared and analyzed using the proposed model-differencing based approach.

Overall, our approach reported pair-wisely seven or eight feature changes between the four major variants in the WFMS product family. We presented some examples of these feature changes in Section 3.1. Our approach reported one wrong feature matching (*SelectByYearOrder*, *SelectByYear*) between $WFMS^{Chongqing}$ and $WFMS^{Shanghai}$. These two fea-

tures are unique features in $WFMS^{Chongqing}$ and $WFMS^{Shanghai}$ respectively. However, due to their description and structural similarities, our approach reported them as the "same" feature being renamed. Furthermore, our approach missed one feature mapping (*WSPayment*, *WebServicePayment*) between $WFMS^{Shanghai}$ and $WFMS^{Shandong}$. Although the two features represent the similar functionality in two product variants, neither their descriptions nor their hierarchical dependencies with other features are similar enough for them to be recognized as a pair of corresponding features.

Before we conducted our case study, we wondered to what extent the name or descriptions of a feature may undergo a systematic renaming e.g. "FudanUniv ->ChongqingUniv". The results showed renaming is just one type of evolution. We did not count what percentage of evolution is related to renaming. Our intuitive observation is that it only takes a small percentage. In addition, those renamed features are usually not related to the names of different product variants.

The system expert of WFMS found the proposed approach useful in three ways. First, although it is possible for him to manually identify the changes to product features in this small-scale product family, the manual analysis would be ad-hoc and require a high familiarity with the subject product family and its evolution history. In contrast, our approach provides a systematic way to assist him in the analysis of feature evolution in a software product family. Second, our approach recovers the traceability of product features across product variants. This helps to understand variants of existing features and adapt "right" features for reuse in new products. Third, the reported feature changes reveal the inconsistencies of feature descriptions and dependencies in product variants. Understanding and reconciling these inconsistencies is the prerequisite to extractive reengineering for a domain model, representing the common and variant features of a software product family.

3.4.2 An empirical study with synthesized PFMs

The WFMS case study demonstrates qualitatively the effectiveness of our approach in understanding feature evolution in a family of product variants. However, we would like to further investigate quantitatively how our method scales up to many product variants characterized by many features that have changed over time. Inspired by Thüm et al's recent work [162], in which the synthesized feature models were utilized to evaluate the classification of feature model evolution, we have also designed a controlled experiment to evaluate the performance of our approach with a large volume of synthesized PFMs. This experiment allows us to better understand the strength and weakness of our approach.

3.4.2.1 The generation of synthesized PFMs

We based our experimentation on the combined feature model of *eShop* [122] and *Home Integration System* (*HIS*) [83] from the feature model repository of S.P.L.O.T [121]. Given this feature model, we developed a tool for generating PFMs in two phases. First, the tool instantiates the feature model to obtain an initial family of PFMs. Next, it iteratively selects a PFM from this family and evolves the PFM by applying the six types of feature changes (as defined in Section 3.3.2). The evolution is performed according to the user-defined intensity (the number of types of changes being applied) and scope (the percentage of features being changed).

a) The selection of feature models.

S.P.LO.T [121] is a benchmark for the research on Software Product Line. It documents a repository of feature models. Feature model [81] is a hierarchy of product features, similar to product feature model. But a PFM represents a concrete product, while a feature model represents all the products in an SPL. Feature model captures the variability among these products in terms of mandatory and optional features, which define the features that must or can be selectively included in a concrete product. It also allows the definition of

AND, OR or XOR constraints among the sub-features of a composite feature. AND indicates that all the sub-features of a composite feature must be included in a product as a whole, while OR or XOR indicates that a subset of all the sub-features or only one sub-feature can be included in a product.

We selected the two largest feature models, namely *eShop* and *HIS*. *eShop* has 286 features in total, among which there are 74 mandatory features, 81 optional features and 131 OR-subfeatures under 39 different parent features. There are no XOR-features in *eShop*. *HIS* has 66 features in total, among which there are 44 mandatory features, 10 optional features and 12 XOR-subfeatures under 6 different parent features. We combined the two feature models, *eShop* and *HIS* into one. One may consider that the resulting feature model represents an artificial system that has *eShop* and *HIS* as its two subsystems.

b) The randomized instantiation of PFMs

Given the combined feature model of *eShop* and *HIS*, our date-generation tool first applies a randomized instantiation strategy to generate a family of PFMs from this feature model by randomly selecting features from the root feature down. The instantiation process takes as input three parameters, the size of the initial product family n (n=10 in this experiment), the probability (α_{OR}) of an optional or an OR-feature to be included in a PFM, and the probability (α_{XOR}) of an alternative feature to be included. In this experiment, we configure α_{OR} and α_{XOR} so that at least 70% of generated PFMs include at least 50% of all the features in the combined feature model of *eShop* and *HIS* [162].

c) The randomized evolution of PFMs

Given an initial family of randomly instantiated PFMs, our data-generation tool then iteratively selects a PFM from this family, evolves it by applying up to six types of feature changes (as defined Section 3.3.2) according to the user-defined evolution strategies, and adds the evolved PFM back to the PFM family. This process continues until the size of the

PFM family reaches the user-defined threshold *K* (*K*=20 in our experiment). This randomized evolution process makes the following assumptions:

- We have studied the feature models listed in S.P.L.O.T repository and found that most feature models define features using phrases, such as "Tuition Fee Payment" in the example given in Figure 3.3. Thus, we utilize WordNet [50] to mimic the renaming of features. We also use the Lucent lib to do some simple preprocess (e.g. stemming and removal of stop words) to the feature name, and feature descriptions. Such preprocess will improve the accuracy of the results.The data-generation tool alters the name of a feature by adding word, removing word, replacing word with synonyms, and reshuffling the order of words.
- The data-generation tool applies only the removal of a leaf feature at a time. The reason is that removing an entire feature subtree usually results in a PFM that has much fewer features than others in the PFM family. This often renders it meaningless to use our approach, since such product variants can be considered as completely different products, while the goal of our work is to detect feature changes in a family of similar product variants.
- The data-generation tool only applies the feature splitting and merging to composite features. Technically, if a leaf feature is split into two leaf features, it is considered as a new leaf feature being added. Similarly, merging two leaf features is considered as removing one of the leaf features.

The evolution of a PFM is performed according to the user-defined intensity (the number of types of changes being applied) and scope (the percentage of features being changed). More specifically, we have designed two evolution strategies: feature-centric and change-type-centric. With the feature-centric strategy, one can specify the percentage of features that will be changed during the evolution process, but the types of changes be-

ing applied to each changed feature are freely chosen. In contrast, with the change-type-centric strategy, one can specify several types of changes that will be applied to the features of a PFM, but each type of changes can be applied to different sets of randomly chosen features. In the next section, we discuss how we apply these two evolution strategies in our experiment.

3.4.2.2 The performance of our technique

The data-generation tool generates a family of PFMs, which are similar but also different from each other. We randomly select S (S=10 in our experiment) pairs of PFMs from this PFM family and apply the approach presented in Section 3.3 to detect the feature changes between them. The data-generation tool records the change history that the PFMs have undergone during the randomized evolution process. This change history serves as an oracle to evaluate the accuracy and robustness of our approach.

Given two PFMs, PFM_1 and PFM_2, we denote the recorded change history (i.e., expected changes) and the detected feature changes by our approach (i.e., reported changes) as *Expected Change-Tuple Set* M_E=$\{<f_1, f_2, changetype>\}$ and *Reported Change-Tuple Set* M_R=$\{<f_1, f_2, changetype>\}$ respectively, where f_1 refers to a feature in PFM_1, f_2 is the corresponding feature of f_1 in PFM_2, and *changetype* is the type of changes between the two features. The *changetype* can be *unchanged*, *rename*, *move*, *split*, or *merge* (See Section 3.3.2 and Section 3.3.4). Since our objective in this experiment is to evaluate the accuracy of our approach in identifying corresponding features in two PFMs, we omit the addition and removal of features. We evaluate the accuracy of our approach in terms of the precision and recall: precision P is the percentage of correctly reported changes, i.e., $|M_R \cap M_E|/|M_R|$ and recall R is the percentage of changes reported, i.e., $|M_R \cap M_E|/|M_E|$.

a) The results of change-type-centric strategy

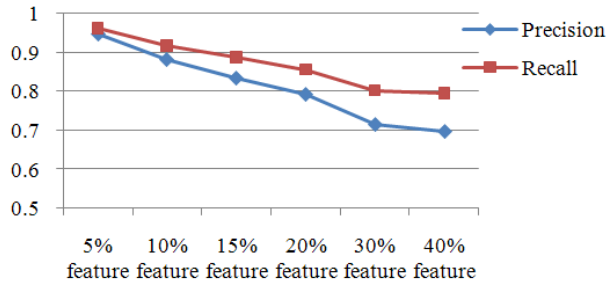

Figure 3.5. The precision and recall for change-type-centric strategy

Figure 3.5 summarizes the precision and recall of our approach in an experiment in which a family of PFMs has been evolved according to change-type-centric strategy. In this experiment, all six types of changes (see Section 3.3.2) have been applied during the evolution process. We incrementally increased the scope, i.e., the percentage of features that will be changed by each type of changes, from 5% to 40%. We compute the precision and recall of our approach in S (S=10 in this experiment) comparisons and then take the arithmetic average value as reported in Figure 3.5.

The accuracy of our approach degrades as the scope of changes increase. It seems that it hits the bottom at the scope of 30%, which is really a very intensive evolution. Since each of six types of changes have been applied to 30% of randomly chosen features, each feature of this PFM statistically undergoes about 1.8 times of different types of changes on average. If such intensive evolution happens to a real-world system, the original product and the resulting product would be deemed as two completely different products.

b) The results of feature-centric strategy

We have conducted another experiment in which a family of PFMs has been evolved according to feature-centric strategy. Figure 3.6 and Figure 3.7 summarizes the results. In this experiment, we first decide the scope of features to be changed and then increase the intensity of changes from one type of change to six types of changes. We run this experiment at four increasing scopes, i.e., 10%, 20%, 30% and 40%. In general, the precision and

recall of our approach drop as the intensity and scope of changes increase. But it still recovers 86% of all the corresponding features at the precision of 81% in the worst scenario, in which 40% of features have been changed, each of which suffers three randomly chosen types of changes.

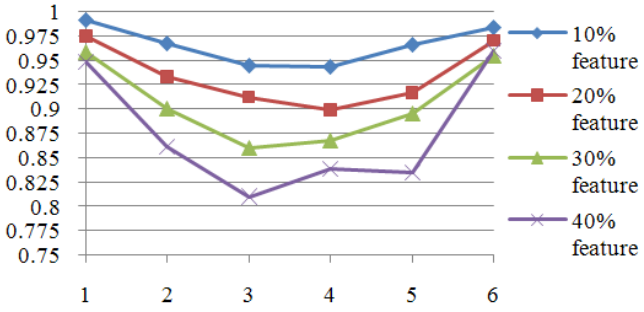

Figure 3.6. The precision for feature-centric strategy

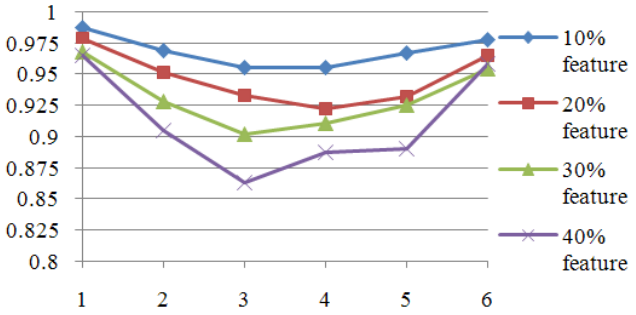

Figure 3.7. The recall for feature-centric strategy

One interesting observation in Figure 3.6 and Figure 3.7 is that when the scope (i.e., the percentage of feature to be changed) is fixed, the precision and recall of our approach by randomly applying five or six types of changes are actually better than that of applying three or four types of changes. We manually inspected the experimental data and found out that this is resulted from the removal of features. At the intensity of five or six types of changes, it is highly likely that the type of feature removal will be applied during the evolution process. When a feature is removed, all the changes that have already been made to it will be lost. Thus, feature removals actually simplify the comparison.

With feature-centric strategy, the scope of changes is fixed, which means that the rest of the features remain unchanged. In this case, even a simple name-based matching approach can recover at least *1-p%* (let the scope of changes be *p%*) of all the corresponding features. We comparatively study our approach against the simple name-based matching approach. Overall, our approach can recover more than 60% of the corresponding features that are missed by simple name-based matching. In a case in which 80% of features have been moved and/or renamed, the name-based matching only reports 27% of all the corresponding features, while the recall of our approach is 67%. This is because our approach identifies corresponding features based on not only their names but also their structural context.

c) Summary and limitation

Overall, our approach is able to produce an accurate change reports between the PFMs of product variants, even the PFMs have undergone intensive evolution. We manually inspected our experimental data and identified two main causes of false positive (i.e., erroneously reported) changes and false negative (i.e., missed) changes using our approach.

First, it is difficult for our approach to determine the correspondences between features with little or very similar structural context (i.e., dependencies and relationships with other features). For example, the leaf features become an issue, since they have no structural information other than their name property. Consider an example from *eShop* system. One product variant has a leaf feature *Internal Tracking*, while the other has a leaf feature *Partner Tracking*. When comparing the PFMs of these two products, our approach reports that *Internal Tracking* and *Partner Tracking* are the "same" feature being renamed in different product variants. However, *Internal Tracking* and *Partner Tracking* are actually two alternative ways of shipment tracking, which should not be considered as the "same" feature.

Second, if product features suffer various types of changes at the same time, for example, a feature *f* is moved to another composite feature, and then split into several features, finally it is renamed, and our approach may not recognize the correspondences between the origin feature *f* and the resulting feature, since their name properties and structure changed dramatically. Since we keep track of the change history of PFMs in the data-generation process, we consider such cases as missed changes. However, a human expert may deem such two features as two different features since they have been changed dramatically, even though one feature is the origin of the other.

3.5 Application

Having evaluated the quality and robustness of our approach, the next question we would like to address is "what is this good for?" In this part, we place the work presented in this chapter in the overall context of our research on extractive reengineering into SPL. We will briefly discuss three applications based on the results of this work. These applications are currently under development – to a different degree of maturity.

The long-term objective of our research is to support reengineering a family of similar product variants into an SPL for systematic reuse. Figure 3.8 depicts the overall methodology we have adopted for our work. The input to our methodology is the software artifacts of product variants at different levels of abstraction, for example, product feature model at requirement level, UML class model at architecture and design level, and Program Dependence Graph (PDG) at implementation level. The output of our reengineering methodology is a collection of core assets of an SPL, which may include domain feature model (FM) [81], Product Line Architecture (PLA) [33] and generic components (GenericComp) [138].

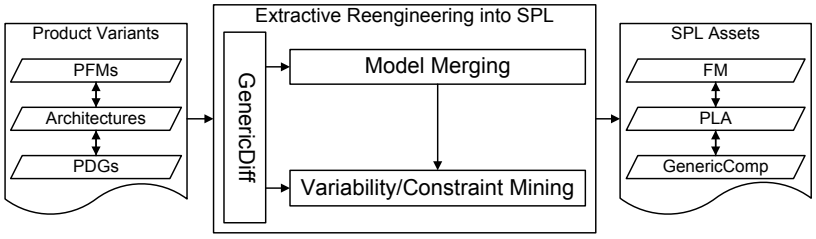

Figure 3.8. Reengineering product variants into SPL

The successful recovery and understanding of commonalities and differences among the artifacts of product variants strides the first step towards our research objective. In this chapter, we presented our application of *GenericDiff* to detect changes to product features at requirement level. In the following work, we have also applied *GenericDiff* to compare software artifacts of product variants at design and implementation level.

Based on the differences *GenericDiff* reports, we are currently investigating the use of model merging and data-mining techniques to reverse-engineer the configurable domain feature model, product line architecture and generic components. For example, we are using description logic [7] to model and reason about the conflicts and inconsistencies among product feature models and investigating the merging of PFMs of product variants using graph transformation tools, such as [147]. In addition to model merging, we are also investigating the recovery of the variability and other general constraints between product features by mining association rules [27] or subtree patterns [31] from the differences among the PFMs of product variants.

Last but not least, recovering traceability in software artifacts of product variants across requirement, design and implementation can provide important insights into the development and maintenance of an SPL. Such traceability is also essential in the derivation of concrete products from an SPL. Researchers have investigated the use of information retrieval [3], scenario-based dynamic analysis [48], or the combination of both [135] for the recovery of traceability. However, existing work on traceability recovery analyzes only

artifacts of a single software system. In our problem setting, we have a family of similar product variants, which should be exploited.

The ability to compare and identify the differences in a family of product variants at different levels of abstraction can assist the task of traceability recovery. The underlying intuition 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. In our ongoing work, we are combining the information retrieval techniques with software differencing results to recovery feature–design element–code fragment traceability in a product variant family.

3.6 Summary

In this work, we presented our approach to understand feature evolution in a family of software product variants. We entail that features and their dependencies for each product variant are documented as product feature model. The innovation of our approach is to exploit model differencing technique (*GenericDiff*) to detect evolutionary changes to product features at requirement level. Based on the differences between product feature models as reported by *GenericDiff*, our approach automatically infers evolutionary changes that occurred to product features of different product variants.

We evaluated the effectiveness and scalability of our method using a real-world product family of financial systems as well as a large volume of systematically synthesized data. We showed that our method yields good results and scales to large systems.

In the current study, we use the term “evolution” to assume that the product feature model may undergo long-term incremental changes. Note that the long-term incremental changes do not mean the great and dynamic changes. For the whole evaluation part, we are testing to what extent our approach can identify the evolutions among PFMs. Certainly, too

great evolutionary changes lead to the huge difference between two PFMs, and it may fail any differencing approach to find the possible evolution changes between these two PFMs.

In the future, based on the results of this work, we plan to investigate merging techniques to support the (semi-)automatic reconciliation of inconsistent product feature models. Furthermore, we also plan to exploit data mining techniques to discover the variability and other general constraints among product features. These techniques will lead to further automation of extractive reengineering of a family of similar product variants into an SPL.

4 Understanding Variability in Implementation of Product Variants

This chapter shares materials with [174], which aims at resolving the problem RQ2 introduced Section 1.1. We introduce the importance of understanding the differences of similar code in the context of product line and refactoring. We present a PDG-based differencing approach on clone instances and evaluate it on two industrial products, namely JavaIO library and Eclipse JDT test library.

4.1 Introduction

Unlike the domain analysis at the requirement level, the analysis at the implementation level is more laborious and error-prone. Discovering the variability in the millions of lines of source code of the products leads to the costly computation. Furthermore, even if it is feasible to directly compare the millions of lines the source code, in what terms should the variability be considered? Should it be considered in terms of textual differences or differences in other representations? Just as the clones are used to identify the aspects [144], clones sometimes indicate the similar features. Based on the clone detection report, the developers are interested to know which features/components recur across product variants, and what differences among similar features/ components are induced by different contexts (see RQ2 in Section 1.1).

However, for the RQ2, the common clone detectors provide little or no additional information to aid developers in understanding the contextual differences among the clone instances. To ease post-detection analysis of clones, researchers have investigated using textual differencing [79,85], code metrics [9,10], visualization [165], and query-based filtering techniques [184]. However, these clone analysis methods analyze only the information of clones themselves, ignoring the program context in which clones occur, and thus cannot identify contextual differences among clones shown in Figure 4.1.

```
//Method FeeInfo.initInfo in WFMS^Fudan
… //clone section same as that in WFMS^Shanghai
1.  for (int i = 0; i < yearTemp.size(); i++) {
2.      boolean ifTS = false;          //if the item is locked by Delegat…
3.      … //clone section same as that in WFMS^Shanghai
4.      rsGetFeeInfo = stmtGetFeeInfo.executeQuery(sqlGetFeeInfo);
5.      if(Global.nTrim(rsGetFeeInfo.getString("bank_mark")).equals("Y"))
ifTS = true;                //if it is locked by Delegat…
6.      … //clone section same as that in WFMS^Shanghai
7.      feeInfo.add(yearFee);
8.      if(ifTS) lockTags.add("Y");   //if it is locked by Delega…
9.      }
```

```
//Method FeeInfo.initInfo in WFMS^Shanghai
… //clone section same as that in WFMS^Fudan
1.    for (int i = 0; i < yearTemp.size(); i++) {
2.        /* don't support Delegation Lock */
3.      … //clone section same as that in WFMS^Fudan
4.      rsGetFeeInfo = stmtGetFeeInfo.executeQuery(sqlGetFeeInfo);
5.
6.      … //clone section same as that in WFMS^Fudan
7.      feeInfo.add(yearFee);
8.      java.sql.Statement stmtLock = conn.createStatement(); /* from this line,
          the code is about Operation Lock*/
9.      String sqlLock = "select sysdate,billtime from lastbill", … // more code
```

Figure 4.1. Differences of two clone fragments

To ease the post-detection analysis, we propose an approach and a tool called CloneDifferentiator. First, we use clone detection tool [11] to find the clone candidates. We capture contextual information of clones from Program Dependence Graphs (PDGs) generated by Wala [204]. These PDGs encode data and control dependencies between program statements. We then use graph matching techniques *GenericDiff* [175] to compute a precise characterization of clones in terms of the structural differences and differential properties between their PDGs, from which several patterns of contextual differences are recognized. The patterns of contextual differences include *Differential Statement, Differential Block, Missing Statement, Missing Block, Missing Branch, PartialMatch Branch*. Then CloneDifferentiator allows developers to formulate queries to distill candidate clones for a given refactoring task, in terms of clones and their contextual differences that developers would like to inspect. It also allows developers to interactively inspect clones and their contextual differences in a GUI.

Let us look at an example. As explained in Section 3.1 that comparison of PFMs may tell multiple possibilities for features like ***DelegationLock*** and ***OperationLock***, Figure 4.1

illustrates two cloned code fragments due to the variant feature *DelegationLock* and *OperationLock* in the method *FeeInfo.initInfo()* from $WFMS^{Fudan}$ and $WFMS^{Shanghai}$ respectively. The basic functionality of *FeeInfo.initInfo()* is to read data from database and initialize a new object of *FeeInfo*. In $WFMS^{Fudan}$, *FeeInfo.initInfo()* further supports the delegation lock for the object of *FeeInfo*; while in $WFMS^{Shanghai}$, *FeeInfo.initInfo()* does not support the delegation lock but the operation lock. As shown at line 2, 5 and 8 of the upper code fragment in Figure 4.1, the delegation lock entails reading the value of field "*bank_mark*" from the database and then proceeds accordingly. Operation lock requires reading data from another table ***lastbill***, as the code shown at line 8, 9 of the lower code fragment in Figure 4.1. Our clone differencing technique reports the differences between two clones as *Missing Statement* at line 2, *Missing Branch* at line 5 and *Differential Block* at line 8. Thus, the results complete the comparison of PFMs and shows that ***DelegationLock*** and ***OperationLock*** are different features rather than the renamed feature.

With the support of CloneDifferentiator, developers no longer need to inspect all clones reported by clone detectors for a given refactoring task. They no longer need to manually explore contextual information of clones and determine their contextual differences. Instead, developers are now directly informed by contextual differences of clones, based on which they can query and filter clones in a task-oriented manner. We evaluated the effectiveness of our approach and the CloneDifferentiator tool in two empirical studies aiming at refactoring JavaIO library and Eclipse JDT-model unit-test suites. Our studies show that CloneDifferentiator is able to distill a small number of useful clones for various refactoring tasks, and thus reduces the effort of post-detection analysis of clones for refactorings.

We make the following contributions in this chapter:

- We identify contextual differences of clones that must be identified and understood for correct reverse engineering and maintenance tasks.

- We present an automated approach to help developers distill useful clones for a given refactoring task by finding and analyzing contextual differences of clones.
- We report two empirical studies and demonstrate the effectiveness of our approach for post-detection analysis of clones for refactorings.

4.2 A Motivating Example in Refactoring

Consider a developer John who would like to refactor Java NewIO library to remove code duplication. One of the specific refactorings that John is interested in is to pull up cloned methods from subclasses into superclass. That is, he is interested in *cloned methods that occur in sibling classes* and *appear to perform same computation*.

John uses CloneMiner [12] for clone detection in Java NewIO. CloneMiner reports 98 clone sets in Java NewIO library; each clone set consists of 2 – 50 cloned methods [184]. John then uses CloneAnalyzer [184] to inspect the detected clones in Java NewIO library. Although CloneAnalyzer provides a rich set of information about the detected clones, it offers little help in identifying candidate clones for his pull-up method refactoring. John has to manually inspect all clones one by one; he resorts to Java Source Compare of Eclipse IDE to determine the differences between cloned methods.

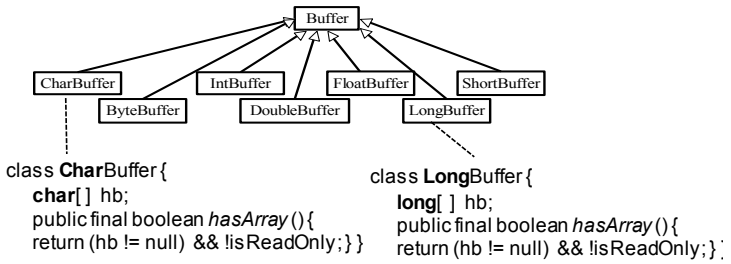

Figure 4.2. Can we pull-up these cloned methods?

After inspecting 69 clone sets, John identifies a clone set of seven cloned methods (see Figure 4.2), which seems to be a good candidate for pull-up method refactoring. The method *hasArray()* is cloned in seven subclasses of the *Buffer* class. At the first glance it may

look that this clone could be removed by pulling up the cloned methods *hasArray()* to the superclass *Buffer*, because seven *hasArray()* methods are *textually identical*. However, when John tries to remove these cloned methods using the Eclipse's Pull Up refactoring, Eclipse reports an error that field *hb* referred in the cloned methods has different data type in different subclasses. For example, as shown in Figure 4.2, the data type of *CharBuffer.hb* is *char[]*, while the type of *LongBuffer.hb* is *long[]*.

Note that for the clarify of illustration we show closely the declaration of field *hb* and the method *hasArray()* that uses *hb*. However, in the source code of the buffer subclasses, the declaration of field *hb* is actually quite far away from (about 660 lines of codes in between) the declaration of method *hasArray()*. Furthermore, Java Source Compare compares programs at textual level and cannot detect type difference of field *hb* of different subclasses. Consequently, John does not notice this important difference among the cloned methods until his attempt to pull-up them into the *Buffer* superclass fails.

The type difference of field *hb* of different buffer subclasses prevents seven cloned methods *hasArray()* from being pulled up into the superclass *Buffer*, because pulling up *hasArray()* requires pulling up field *hb* at the same time. Unless John finds a way to properly deal with type difference of field *hb* (for example using generic type), pulling up *hb* into the superclass *Buffer* would lead to errors in other parts of buffer subclasses.

4.3 Contextual Analysis of Clones

The examples in Section 4.1 and 4.2 illustrate *contextual analysis* that developers have to do when performing refactorings affects clones. Our study suggests that in order to correctly understand clones before performing refactorings developers must examine several pieces of *contextual information* of clones, including program elements referenced in the cloned methods (e.g. fields being accessed in our motivating example), associated proper-

ties of these program elements (e.g. data type of the field), and control and data flow surrounding the cloned code fragments.

Differences in the contextual information of clones may affect computation performed by clones, and thus must be identified and understood. The type difference between field *hb* of different buffer subclasses is a simple example of what we call *differential statements* among seemingly similar (or even identical) code clones, i.e. statements that appear in similar control and data flow context in the cloned methods, but may perform different computation.

```
PipedOutputStream (Java IO 1.5)
36.   private PipedInputStream sink;
101.  public void write(int b) throws IOException {
102.    if(this.sink == null)
103.      throw new IOExcetion("...");
104.      this.sink.receive(b); }
PipedWriter (Java IO 1.5)
25.   private PipedReader sink;
103.  public void write(int c) throws IOException {
104.    if(this.sink == null)
105.      throw new IOException("...");
106.      this.sink.receive(c); }
```

Figure 4.3. Differential statements

Figure 4.3 presents another example of differential statements from Java IO library. The overall control and data flow of the cloned methods are identical, but the two methods perform difference computation. This is because the two methods read different fields, *PipedOutputStream.sink* versus *PipedWriter.sink*, and the type of the two fields are also different, *PipedInputStream* versus *PipedReader*. Furthermore, the two methods invoke different methods, *PipedInputStream.receive()* versus *PipedReader.receive()*. In fact, many clones in Java IO library have such differential statements, which are resulted from processing byte data and char data respectively. Note that examining only the cloned methods themselves cannot surface these *differential statements*, because the two methods are textually identical.

```
ObjectInputStream.read(byte[] buf, int off, int len)
806.  if(buf==null) {
807.    throw new NullPointerException("...");}
809.  int endoff=off+len;
810.  if(off<0||len<0||endoff>buf.length || endoff<0) {
811.    throw new IndexOutOfBoundsException();}
ObjectInputStream.readFully(byte[] buf, int off, int len)
976.  int endoff=off+len;
977.  if(off<0||len<0||endoff>buf.length || endoff<0) {
978.    throw new IndexOutOfBoundsException();}
```

Figure 4.4. Missing branch and statements

Two other important types of contextual differences of clones are *missing statements* (i.e. statements that appear in some cloned methods but not others) and *missing/partially-matched branches* (i.e. missing or inconsistent branches among cloned methods). Figure 4.4 shows an example. The method *read()* checks if *buf* is *null*, creates and throws a *Null-PointerException* if *buf* is *null*, while *readFully()* does not have such explicit checking. Instead, *readFully()* implicitly throws a *NullPointerException* when deferencing a null *buf*. This example shows a common inconsistent program style in Java IO for validating input parameters and handling exceptions. Clearly, the method *read()*represents a better solution as it will fail as quickly as possible in the case of a null buffer. Furthermore, it allows em-bedding a program-specific error message in the thrown exception.

Once these contextual differences are identified, they can help distill useful clones for a given refactoring task. For example, developers who are interested in pulling-up cloned methods may formulate a query searching for cloned methods that are in sibling classes and have zero contextual differences. Note that the cloned methods shown in Figure 4.2 and Figure 4.3 will not be returned by the query, because they have differential statements, even though they look identical. But they will be returned by the query searching for cloned method that can be replaced by generic methods. As another example, developers who are interested in reconcile small discrepancies among cloned methods may formulate a query searching for clones such as the one shown in Figure 4.4. They can remove the

differences resulted from inconsistent program styles and then extract parameter-validity-checking logic into a utility method.

Clearly, given large numbers of clones, manual contextual analysis of clones is impractical. Now the question becomes: *how can we precisely capture contextual information of clones and automatically identify contextual differences of clones*?

4.4 The Approach

We propose an automatic approach and tool called CloneDifferentiator that can help developers identify and analyze contextual differences of clones. In this section, we first give an overview of our CloneDifferentiator approach. And then we discuss how CloneDifferentiator represents contextual information of clones, and how it computes contextual differences of clones and what types of differences it reports.

4.4.1 Overview

CloneDifferentiator analyzes cloned methods detected by one of the existing code clone detectors. Clone detectors usually group cloned methods to form clone sets. The cloned methods called clone instances in each clone set are pair-wise similar to each other, according to similarity metrics used by the clone detector. Our CloneDifferentiator tool currently uses CloneMiner [11] for clone detection. It can detect the high level similarity, including structural clones and gapped clones. We used the default setting up of CloneMiner (e.g. min-token length 30 for a clone). We used our approach to compare those clone method which contains the gapped clone. Actually, the structural clones reported by CloneMiner are generally more complicated than those reported by CCFinder [80] or CloneDR [20]. These complicated clones allow us to see the performance and limitation of our approach. However, it is important to note that our approach does not make any specific assumptions regarding clone detectors.

CloneDifferentiator raises the level of analysis of clones to Program Dependency Graph (PDG). It represents the contextual information of cloned methods using PDG. PDG allows CloneDifferentiator to precisely capture the contextual information of clones, including program elements being referenced in cloned methods, associated properties of these program elements, and data and control flow information in cloned methods.

Given the PDGs of the cloned methods in a clone set, CloneDifferentiator then uses graph differencing algorithm to compare PDGs of clones. It automatically detects contextual differences of clones, in terms of differential statements or blocks, missing statements or blocks, and missing or partially-matched branches. These contextual differences can be used to formulate queries for distilling useful clones for a given maintenance task.

Our CloneDifferentiator tool [173,178] stores its contextual analysis results of clones in a relational database. Stored information includes PDGs of cloned methods and the instances of different types of contextual differences of these clones. CloneDifferentiator is equipped with a set of simple filters for filtering clones based on the types and number of their contextual differences. Furthermore, it allows the developer to formulate task-specific queries in terms of which clones and what types of their contextual differences he would like to inspect.

Finally, our CloneDifferentiator tool implements three views that allow developers to interactively inspect cloned methods and their contextual differences: *CloneDiff TreeView* that summarizes cloned methods being analyzed and their contextual differences; *PDG Viewer* for graphically inspecting clones and their differences in PDG; *CloneDiff Compare Editor* for inspecting clones and their differences in an enhanced Eclipse compare editor.

4.4.2 Representing contextual information of clones as PDG

Let us first discuss why we adopt PDG to represent contextual information of clones. We then describe the PDG representation that our current CloneDifferentiator tool adopts for contextual analysis of clones.

4.4.2.1 Why PDG?

A Program Dependence Graph (PDG) [35,52] is a static representation of the control and data flow through a program. It is commonly used for program optimization and slicing. CloneDifferentiator adopts PDG as the representation of contextual information of clones, and computes contextual differences of clones at PDG level. This allows it to provide a precise characterization of contextual differences of clones than lexical or syntactic differencing techniques, because PDG abstracts away many textual and syntactic differences in contextual analysis of clones.

Let us look at an example from our empirical studies of clones in Java IO library using CloneDifferentiator. Figure 4.5 presents the contextual differences of a pair of cloned methods *listFiles(FilenameFilter)* and *listFiles(FileFilter)* in a *CloneDiff Compare Editor* (see Section 4.4.4).

CloneDifferentiator reports that the cloned code fragments (inside light-grey box) of the two methods have a pair of *differential parameters* (highlighted in light blue background). The two parameters declare different data types (*FilenameFilter* versus *FileFilter*). CloneDifferentiator also reports that the two cloned methods have a pair of *differential method invocations* (*FilenameFilter.accept() versus FileFilter accept()*, highlighted in red background). Furthermore, the method *listFiles(Filenamefilter)* has an additional array-access statement (*ss[i]*, highlighted in yellow double underline and italic font), i.e. a *missing array-access statement* that *listFiles(FileFilter)* does not have.

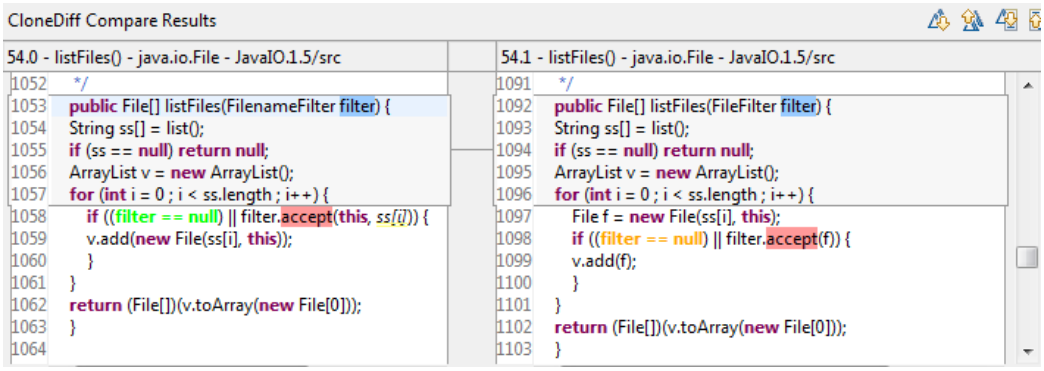

Figure 4.5. Inspecting contextual differences in CloneDiff Compare Editor

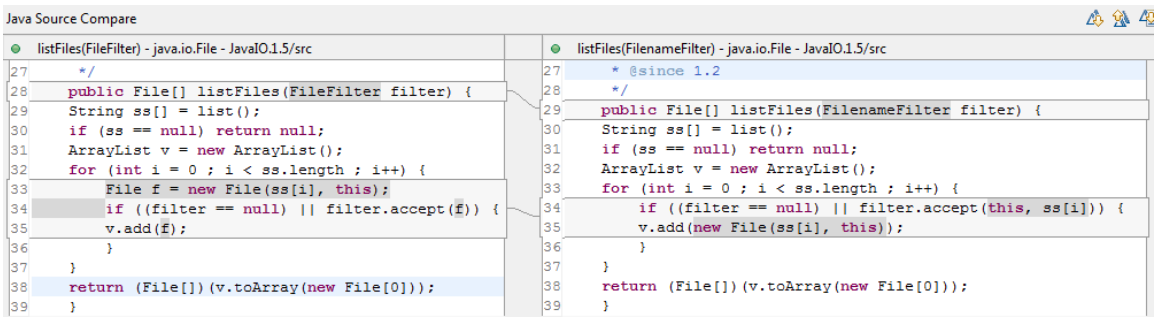

Figure 4.6. Textual differences in Java Source Compare

CloneDifferentiator also detects the difference in the control flow of the two methods: the program control flows from the matched branch statement (*i<ss.length*) directly to the unmatched branch (*filter==null*) in *listFiles(Filename-Filter*), while in *listFile(FileFilter*) the control flows first to the instantiation of a *File* object and then to the unmatched branch (*filter==null*). CloneDifferentiator reports this difference as *partially-matched branches filter==null* (highlighted in green and accent font in the two methods respectively). Note that it is more straightforward to inspect contextual differences resulted from control/data flow differences in *PDG Viewer* (see this example in Figure 4.13).

Compared with the textual differencing results of the two cloned methods (see Figure 4.6), the contextual differences that CloneDifferentiator reports is clearly much more precise. The textual differencing reports some textual differences between the code block (line

33–35) of *listFiles(FileFilter)* and the code block (line 34–35) of *listFiles(FilenameFilter)*. In contrast, such differences are abstracted away in PDGs and thus are not reported by CloneDifferentiator. On the other hand, because textual differencing compares clones as lines and chars, it cannot report the subtle difference in the branch statements *filter==null* and the difference in the actual methods being invoked *filter.accept()*, as reported by CloneDifferentiator.

More importantly, the contextual differences of CloneDifferentiator can be directly queried in a task-oriented manner. In contrast, because textual differences ignore semantic information to which they correspond, they have to be manually examined and interpreted. For example, textual differencing identifies the differences between *FileFilter* (line 28 of *listFiles(FileFilter))* and *FilenameFilter* (line 29 of *listFiles(FilenameFilter))*. However, textual differencing cannot automatically determine that the detected textual difference is due to the type difference of two parameters.

Syntactic differencing techniques, such as change distilling [53] that compares Abstract Syntax Tree (AST), are more robust than textual differencing, for example they can detect the type difference of the parameters of the two *listFiles* methods. However, syntactic differencing is still sensitive to the arbitrary syntactic decisions a developer made while developing a program. For example, statements *File f = new File(); v.add(f)* and *v.add(new File())* result in different ASTs, but they yields the same PDG. Furthermore, AST-based differencing techniques are agnostic of control and data flow through a program, and thus cannot detect contextual differences resulted from control and data flow, such as the differences between the two *filter==null* statements in the two *listFiles* methods.

4.4.2.2 Wala PDG

In general, the nodes of a PDG consist of three categories of program statements constructed from the source code: simple statement, expressions, and control points. A control

point represents a point at which a program branches, loops, enters or exits a procedure. The edges of a PDG encode the data and control dependencies between program statements.

Our CloneDifferentiator tool uses Wala [204] to generate PDGs of cloned methods. Wala is a static analysis library for Java. It is important to note that using Wala for PDG generation in our CloneDifferentiator tool is only an implementation choice because Wala is open source and publicly available. Furthermore, we conducted empirical studies on Java software systems. Our CloneDifferentiator approach is not limited to any specific PDG generation tools, nor is it limited to analyzing clones in Java software systems.

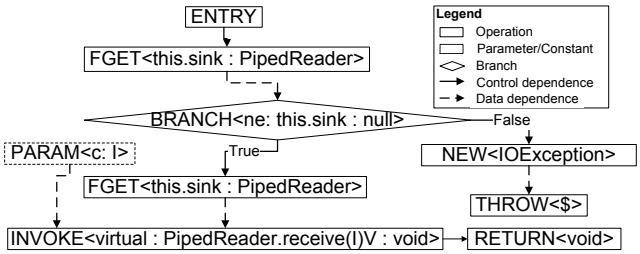

Figure 4.7. Wala-PDG example: *PipedWriter.write(int):void*

Given a Java software, Wala generates a system dependency graph that is composed of PDGs of all methods in the system and the inter-procedure dependencies of these PDGs. Because we are only interested in cloned methods, CloneDifferentiator outputs the PDG of each cloned method for the subsequent contextual differencing and analysis. Wala-PDG captures *contextual information* of cloned methods, including program elements referenced in cloned methods, associated properties of these program elements, and control and data flow in cloned methods.

Wala-PDG supports three categories of bytecode-like program statements constructed from source code: simple statement, control point, and expression. Simple statement includes field read/write *FGET/FPUT*, method invocation *INVOKE*, unary and binary opera-

tion (*negate*, *add*, *minus*, *multiply*), compare statement (>, <, !=), array load/store *ARRAY-LOAD/ARRAYSTORE*, type checking *INSTANCEOF*, type casting *CAST*, object creation *NEW*, and exception throwing *THROW*. Control points include branching *BRANCH* and switching *SWITCH*. Expressions include parameter *PARAM* and constant *CONST*. Different types of program statements consist of different sets of properties, such as identifier, data type, operator code, and operand. Figure 4.7 shows a partial Wala-PDG for the cloned method *PipedWriter.write(int):void* listed in Figure 4.3.

4.4.3 Detecting contextual differences of clones by PDG differencing

Given a clone set of n cloned methods $\{m_1,\ldots,m_n\}$, let PDG_i and PDG_j be the PDGs of cloned methods m_i and m_j $(i \neq j; 1 \leq i,j \leq n)$, CloneDifferentiator applies graph differencing algorithm to pair-wisely compare PDG_i and PDG_j and detect the contextual differences between the cloned methods m_i and m_j.

CloneDifferentiator also uses GenericDiff [175] (a configurable graph matching framework) for comparing Wala-PDGs. Inspired by [140], GenericDiff reports a domain independent symmetric difference between two input PDG graphs: a set of matched graph nodes and edges that exist in both graphs, and two sets of unmatched graph nodes and edges that exist only one of the two graphs. CloneDifferentiator interpret the GenericDiff's PDG differencing results in terms of meaningful contextual differences of code clones: differential statements or blocks, missing statements or blocks, and missing or partially-matched branches. Interested readers are referred to Section 2.3.2 and GenericDiff [175] for the details about we configure GenericDiff for comparing PDGs.

In the rest of this part, we discuss these contextual differences and their impacts on the contextual analysis of code clones. We illustrate our discussion with examples from our empirical studies.

4.4.3.1 Differential statements or blocks

Differential statements (blocks) represent a pair of Wala-PDG statements (blocks of statements), one from each cloned method, that appear in similar control and data flow context in the two cloned methods, but may perform different computation.

Differential statements are resulted from the following three reasons:

- The *methods being invoked* in method invocation (*INVOKE*) statements and the *fields being accessed* in field access (*FGET*/*FPUT*) statements can be different. Such differences reveal the use of different methods and fields for the same or similar purpose in cloned methods.
- The *operator-code* of method invocation, binary operation, compare, and branch statements can be different. Furthermore, the *data type* of field-read/write, method invocation, type checking, type casting, object creation, array-load/store, and parameter statements can be different. Different operator-codes or data types can result in different program behavior.
- The *operand* of binary-operation, compare, and branch elements or the *value* of constant elements can be different. Different operands or values can affect the evaluation of dependent statements.

Figure 4.8 presents an example of differential statements from our empirical study on JavaIO library. The overall control and data flow of the cloned methods *readArray()* and *readOrdinaryObject()* is similar. But CloneDifferentiator detects three pairs of differential statements between the two methods: the method being invoked in the two method-invocation statements is different: *Array.newInstance()* versus *ObjectStreamClass.newInstance()*; the *operator-code* of these two method invocations is different: *static* versus *virtual*; the constant *operand* of the branch statements is different: *TC_ARRAY* versus *TC_OBJECT*.

```
ObjectInputStream.readArray(boolean)
1581.    ……
1582.    if(bin.readByte() != TC_ARAY)
1583.       throw new StreamCoruptedException()
1592.    array = Array.newInstance(ccl, len)
1593.    ……
ObjectInputStream.readOrdinaryObject(boolean)
1689.    ……
1690.    if(bin.readByte() != TC_OBJECT)
1691.       throw new StreamCoruptedException()
1698.    obj = desc.isInstantiable() ? desc.newInstance() …;
1699.    ……
```

Figure 4.8. Differential statements

Differentiator blocks are similar to differential statements; the only difference is that a block consists of a sequence of statements, i.e. a subgraph of the PDG of the cloned methods.

```
CreateMemberTests.test002()
70.  ……
71.  compilationUnit = getCompilationUnit(…, "E.java");
76.  IField sibling = type.getField("j");
77.  type.createField("int i;", sibling, true, null);
78.  ……
CreateMemberTests.test003()
89.  ……
90.  compilationUnit = getCompilationUnit(…, "Anno.java");
95.  IMethod sibling =type.getMethod("foo", new String[]{});
96.  type.createMethod("String bar()", sibling, true, null);
97.  ……
```

Figure 4.9. Differential block

Figure 4.9 presents an example of the two cloned test methods from our empirical study on Eclipse JDT unit tests. Both test methods are used to test creating a member in a class. However, CloneDifferentiator detects that the cloned methods have a pair of differential blocks, which reveal the differences of the two test methods in retrieving and creating different types of class members. That is, *test002()* tests creating a field, while *test003()* tests creating a method.

4.4.3.2 Missing statements or blocks

Missing statements (blocks) represent statements (blocks of statements) that appear only in one of the cloned methods but not the other.

```
ObjectInputStream$BlockDataInputStream.peek()
3960.    if(blkmode) {
3961.      return (end>=0) ? (buf[pos] & 0xFF) : -1;
3962.    } else { ... }
ObjectInputStream$BlockDataInputStream.read()
3963.    if(blkmode) {
3964.      return (end>=0) ? (buf[pos++] & 0xFF) : -1;
3965.    } else { ... }
```

Figure 4.10. Missing statements

Figure 4.10 presents an example of missing statement between the two cloned methods. CloneDifferentiator detects that one of the cloned method *read()* has two additional statements (line 3964) that the method *peek()* does not have: a binary operation that adds *pos* by 1 and a field-write statement that updates *pos* with the new value. These two missing statements reveal the key differences between *read()* and *peek()*: *read()* consumes the value, while *peek*() only returns the value in the stream buffer.

```
SequenceInputStream. read (byte b[], int off, int len)
181.  ... //clone part
187.  else if (len == 0) {
188.        return 0;
189.  }
191.   int n = in.read(b, off, len);
192.     if (n <= 0) {
193.       nextStream();
194.       return read(b, off, len);
195.     }
196.  return n;
PipedOutputStream. write(byte b[], int off, int len)
122.  ... //clone part
129.  else if (len == 0) {
130.        return 0;
131.  }
132.  sink.receive(b, off, len);
```

Figure 4.11. Missing block

Figure 4.11 shows an example of missing statements and block between the two cloned methods. CloneDifferentiator detects that one of the cloned methods *SequenceInputStream.read()* has an additional block that the method *PipedOutputStream.write()* does not have, while *PipedOutputStream.write()* has some statements that *SequenceInputStream.read()* does not have. The two methods share the logic of validating input parameters so that they are reported as cloned method by CloneMiner. However, the compu-

tation of the two methods is in fact very different as indicated by missing statements and block.

4.4.3.3 Missing or partially matched branches

Missing branches represent control points that appear only in one of the cloned method but not the other, while partially matched branches reveal the inconsistencies between a sequence of control points between the cloned methods.

```
ByteArrayInputStream.read(byte[] buf, int off, int len)
160.  ......
161.  } else if(off<0||len<0||len>buf.length-off) {
162.     throw new IndexOutOfBoundsException();}}
163.  ......
StringBufferInputStream.read(byte[] buf, int off, int len)
95.   ......
96.   } else if(off<0||len<0||
97.          off+len>buf.length||off+len<0) {
98.     throw new IndexOutOfBoundsException();}}
99.   ......
```

Figure 4.12. Partially-matched branches

Figure 4.4 presents an example of a common inconsistent program style in JavaIO that results in missing branch and statements between cloned methods. Figure 4.12 presents a typical example of another common inconsistent program style in Java IO that results in partially-matched branches when checking the validity of input parameters. In this example, both cloned methods perform a sequence of validity checks of parameters *off* and *len* (line 160 versus lines 95/96). CloneDifferentiator detects that the cloned methods have partially-matched branches. These differences reveal similar but also different parameter validity checking of the two methods. The two methods have two pairs of matched branch statements (*off<0* and *len<0*), but they check different expressions (*len>buf.length-off* versus *off+len>buf.length*) to ensure that the sum of *off* and *len* is less than the length of *buf*. Furthermore, *StringBuffer-InputStream.read()* has one more checking (*off+len<0*), which is unnecessary, since it always evaluates to false.

4.4.4 Tool Support

Let us now discuss querying and visualization support that our CloneDifferentiator tool provides for distilling and inspecting clones and their contextual differences.

Our CloneDifferentiator tool is equipped with a set of simple filters for filtering clones based on the types and numbers of their contextual differences. For example, one can easily find clones that have zero contextual difference or clones that have only differential constant statements with different values. Such clones are duplicated codes that can be removed by refactorings [54], such as pull up method, parameterize method. The filters are backed up by the query language for the contextual differences or difference pattern, which is SQL. We store the clone code snippets, the PDGs of clone instances, and also the clone comparison results in the relational database. We then use SQL to express queries for the database.

If the simple filters that CloneDifferentiator provides cannot meet the needs of developers for a refactoring task, CloneDifferentiator allows developers to formulate their own queries (SQL queries in current implementation), using contextual differences of clones as basic building blocks, for distilling candidate clones that are useful for the specific task at hand. Section 4.5.2 and 4.5.3 will review a few such queries for various refactoring tasks.

In addition to filtering and querying clones and their contextual differences, our CloneDifferentiator tool supports the following three views, i.e. *CloneDiff TreeView, PDG Viewer,* and *Compare Editor*, for presenting and visually inspecting clones and their contextual differences.

CloneDifferentiator lists clone sets returned by a query in *CloneDiff TreeView*. Clone sets can be easily navigated to through the tree. For each pair of cloned methods in a clone set, *CloneDiff TreeView* summarizes the types and number of contextual differences that

this clone pair has. Clone sets can be sorted by the signature of cloned methods and/or the types and number of their contextual differences.

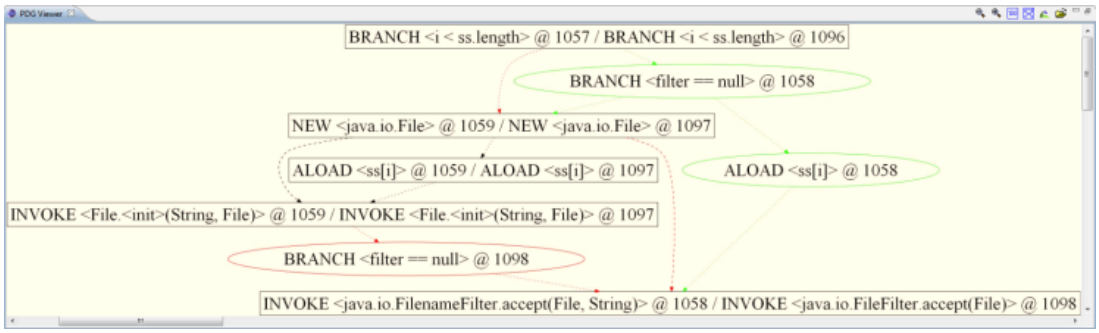

Figure 4.13. PDG Viewer

Double-clicking a clone pair in *CloneDiff TreeView* opens a *PDG Viewer* that visualizes the unified PDG of the selected pair of cloned methods. *PDG Viewer* allows developers to graphically inspect PDGs of clones and their contextual differences, especially differences of data and control flow. For example, Figure 4.13 shows partially the unified PDG of the cloned methods *listFiles(FilenameFilter)* and *listFiles(FileFilter)*. Black nodes/edges represent matched PDG statements and dependences that exist in both methods, while green and red nodes/edges represent unmatched statements and dependences that exist only in one of the two methods. We can clearly see the control- and data-flow differences between the two methods, which results in partially-matched branches *filter==null*.

Although *PDG Viewer* provides an intuitive means to inspect contextual differences of clones, it often becomes difficult to inspect contextual differences in large PDGs. Furthermore, developers are most familiar with examining program differences in compare editor that present programs to be compared side by side. For that our CloneDifferentiator tool extends Eclipse compare editor into *CloneDiff Compare Editor* (see Figure 4.5). This editor displays the source code of the cloned methods side by side; it highlights code segments based on the detected contextual differences of the cloned methods, using different

fonts, backgrounds or underlines. Furthermore, *CloneDiff Compare Editor* can be configured to highlight only specific type(s) of differences of interest to the user. Hovering mouse over a highlighted code segment pops up a short description that summarizes the contextual differences in the hovered code segment.

Note that in this chapter we present clone examples in code snippets and highlight their contextual differences using grey background and bold font. Furthermore, when presenting examples of a type of contextual differences, we often highlight only differences of the specific type being discussed, ignore differences of other types. For example, as shown in Figure 4.8, the cloned method *readOrdinaryObject()* has several statements and control points (for example method invocation *isInstantiable()*, and branch *desc.isInstantiable() ? ... : ...*) that *readArray()* does not have. However, we do not highlight these missing statements and branches in Figure 4.8. This in fact simulates the configurable presentation of *CloneDiff Compare Editor*.

Finally, as discussed above, our experience suggests that it is often more straightforward to visualize and inspect contextual differences of clones in *CloneDiff Compare Editor* than in *PDG viewer*. Thus, in this chapter we present and discuss clone examples using code snippets of cloned methods instead of their corresponding PDGs. However, it is important to note that the highlighted code segments are based on the differencing results of PDGs of cloned methods, instead of simply textual differencing.

4.5 Evaluation

We evaluated our CloneDifferentiator approach and tool on two Java software systems: JavaIO library [203] and Eclipse JDT-model unit tests [193]. JavaIO library 1.5 contains 101 classes and 1038 methods. CloneDifferentiator uses CloneMiner for clone detection. CloneMiner detects 103 clone sets; each set consists of 2 – 15 cloned methods. JDT-model unit tests (jdt.core.tests.model) 3.6.1 contain 336 test suites and 10740 test methods. For

JDT-model unit tests, CloneMiner detects 961 clone sets; each set consists of 2 – 35 cloned methods.

We then use CloneDifferentiator to identify and analyze contextual differences of these detected clones. Our evaluation aims at gaining insights into the following research questions: How often is contextual information of clones different? Do contextual differences of clones in different systems manifest different characteristics? Can contextual differences of clones help to distill useful clones in a task-oriented manner?

In this section, we first analyze the statistics and characteristics of contextual differences of clones that CloneDifferentiator identifies in the clones of the two subject systems (Section 4.5.2 and 4.5.3). And then, we demonstrate how we formulate queries of contextual differences of clones to distill candidate clones that are useful for various refactoring tasks aiming at reducing code duplication in the two subject systems (Section 4.5.2 and 4.5.3).

4.5.1 Characteristics of contextual differences of clones

Our results suggest that in both JavaIO library and JDT-model unit tests the detected cloned methods usually have various types and instances of contextual differences. The differences are usually subtle. The contextual differences of cloned methods of different systems may manifest different characteristics, due to different nature of the systems that results in clones.

Table 4.1 reports the statistics of contextual differences of cloned methods in JavaIO. Each row in the table represents a type of contextual difference discussed in Section 4.4.3. Column "#diff" lists the number of instances of a particular type of contextual difference; column "#cloneset(cc)" lists the number of clone sets that have at least one instance of a particular type of contextual difference; column (#diff/#cc) lists the average number of instances of different types of contextual differences per clone set.

Table 4.1. Statistics of contextual differences in JavaIO 1.5

Type	# diff	#cloneset(cc)	#diff/#cc
Differential Statemt	329	79	4.2
Differential Block[2]	13	10	1.3
Missing Statement	392	80	4.9
Missing Block	68	44	1.6
Missing Branch	26	18	1.5
PartialMatch Brch	21	17	1.2

Table 4.2. Statistics of contextual differences in JDT-model tests

Type	#diff	#cloneset(cc)	#diff/#cc
Differential Statemt	7900	931	8.5
Differential Block	101	90	1.1
Missing Statement	6761	666	10.2
Missing Block	1217	460	2.64
Missing Branch	512	203	2.5
PartialMatch Brch	13	12	1.1

For example, the first row of Table 4.1 shows that CloneDifferentiator identifies 329 instances of differential statements in 109 clone sets; on average one clone set has 3.0 instances of differential statements. Note that one clone set can have more than one type of contextual differences. Therefore, the sum of column "#cloneset" is greater than the number of clone sets that CloneMiner reports.

CloneDifferentiator reports in total 849 (sum(#diff)) instances of different types of contextual differences in the cloned methods of JavaIO library. For a particular type of contextual difference, each clone set has on average at least one instance (#diff/#cc) of that type of difference, for example 1.2 instances of partially-matched branches per clone set. Each clone set has on average three (sum(#cc)/103) types and eight (sum(#diff)/103) instances of contextual differences.

---

[2] In this evaluation, we consider a block that contains at least 6 unmatched statements as unmatched block. Such unmatched blocks usually span at least two lines of code, which is the minimum length of code fragments that clone detectors usually deal with [80].

The most common type of contextual differences of the cloned methods of JavaIO are missing statements (392 instances), followed by differential statements (329 instances). These two types of contextual differences account for about 85% of all 849 instances of differences. Missing blocks (68 instances) and differential blocks (13 instances) account for about 10% of all 849 instances. Partially-matched branches (PartialMatch Brch, 21 instances) and missing branches (26 instances) account for a very small percentage (5%) of all 849 instances. Overall, the differences between cloned methods of JavaIO are usually subtle, but sometimes the differences are significant.

Table 4.2 presents the statistics of contextual differences of cloned methods in JDT-model unit tests. The cloned methods of JDT-model unit tests have much more (sum(#diff)=16504) instances of contextual differences. This is not surprising because JDT-model-unit-tests is a much bigger project and it has nine times more clone sets than JavaIO. However, the percentages of different types of contextual differences in the cloned methods of JDT-model-unit-tests are roughly similar to those of JavaIO. Furthermore, the percentages of clone sets that have a particular type of contextual differences are also roughly similar to those of JavaIO.

One important difference is that the clone methods of JDT-model unit tests has on average more types (sum(#cc)/961) and instances (sum(#diff)/961, #diff/#cc) of contextual differences than the cloned methods of JavaIO. This is mainly because the JDT-model unit test methods are usually longer than the methods of JavaIO.

The other difference is that almost all clone sets of JDT-model unit tests have differential statements (931/961, 96.8%). This is due to the existence of a large amount of differential constant statements. In fact, this reflects a common practice in writing unit tests in which similar tests are developed to test different input values (see examples in Section 4.5.2 and 4.5.3).

4.5.2 Refactoring JavaIO library

Let us now discuss our study on clone-based refactoring of JavaIO library. In this study, we are interested in identifying clones that can be refactored using *Folwer's refactorings* (e.g. *extract method, pull up method*) or *Java generics*, thus reducing code duplication.

4.5.2.1 Refactoring clones using Folwer's refactorings

Many Folwer's refactorings are concerned with code duplication [54]. Folwer's refactorings usually target at identical or almost identical cloned methods, which can be removed by refactorings such as *extract method*, *pull up method*.

To identify candidate clones for Folwer's refactorings, we formulate the following two queries searching for:

1. The cloned methods that have no contextual differences, i.e. the PDGs of such cloned methods are perfectly matched;
2. One of the cloned methods is "part of" the other cloned methods, i.e., the PDG of one cloned methods is the subgraph of the PDG of the other. That is, only one of the clone methods has missing statements, blocks, and/or branches.

The first query returns three pairs of cloned methods. For example, Figure 4.14 presents one pair of these cloned methods. The two methods perform identical computation. Both methods write the eight lower-order bits of the input argument to the output stream and ignore the 24 high-order bits. These cloned methods can be refactored by replacing the body of one method with a call to the other method. Note that CloneDifferentiator does not report that the cloned methods in Figure 4.14 have differential parameters, even though the identifier of the parameters is different (*b* vs. *v*). This is because the two parameters declare the same data type (*int*), and the simple identifier difference does not affect the computation performed by the cloned methods.

```
ObjectOutputStream$BlockDataOutputStream (Java IO 1.5)
1767.   public void write(int b) throws IOException {
1768.     if(this.pos >= MAX_BLOCK_SIZE)
1769.       this.drain();
1770.     this.buf[pos++]= (byte)b; }
ObjectOutputStream$BlockDataOutputStream (Java IO 1.5)
1874.   public void writeByte(int v) throws IOException {
1875.     if(this.pos >= MAX_BLOCK_SIZE)
1876.       this.drain();
1877.     this.buf[pos++]= (byte)v; }
```

Figure 4.14. Cloned methods that have no contextual diffs

The second query returns one pair of cloned methods, *PipedInputStream.checkStateForReceive()* and *PipedReader. receive(int)*. Both *PipedInputStream* and *PipedReader* need to perform the same checking of pipe state in several places before starting receiving data. The developer of *PipedInputStream* recognized the repetition of this state checking and extracted the state checking logic into the method *PipedInputStream.checkStateForReceive()*. In contrast, the developer of *PipedReader* did not extract the state checking logic from *PipedReader.receive(int)* into a separate method. As a result, CloneDifferentiator detects that the state checking method *PipedInputStream.checkState-ForReceive()* is "part of" the method *PipedReader. receive(int)*. Identifying this "part of" relation between the cloned methods suggests the opportunity to extract method.

Overall, only a very few cloned methods in JavaIO library represents identical or almost identical code clones that can be removed by Folwer's refactorings.

4.5.2.2 Relaxed queries for Folwer's refactorings

To identify more candidate clones for Folwer's refactorings, we relaxed the two queries given in the last section, by allowing the cloned methods to have a small number of contextual differences. In particular, relaxed queries allow the cloned methods to contain a maximum of six instances of differential statements, missing statements, missing branches, and/or partially-matched branches.

The relaxed queries return 21 more pairs of cloned methods. Four pairs of these cloned methods have *only differential operator-code and/or operand statements*. For example, the cloned methods *LineNumberInputStream.-read(byte,int,int)* and *InputStream.read(byte,int,int)* are all the same but a pair of *differential operator-code* (*special* vs. *virtual*) method invocations (*LineNumberInputStream.read()* versus *InputStream.read()*). The class *InputStream* declares a template method [57] *read(byte,int,int)* that define the skeleton of reading bytes from the input stream. *InputStream.read(byte,int,int)* calls the abstract method *InputStream.read()*, and the subclasses of *InputStream* (e.g., *LineNumberInputStream*) must implement the abstract method *InputStream.read()* to read the next byte of data from a specific type of input stream. However, the subclass *LineNumberInputStream* duplicates the template method *read(byte,int,int)*, which deviates from the intent of Template Method [57]. Clearly, this duplicate *LineNumberInput-Stream.read(byte,int,int)* should be removed.

The remaining 17 pairs of cloned methods reveal two types of inconsistent program styles in JavaIO library. These inconsistent programming styles result in certain amount of missing statements, missing branches and/or partially-matched branches in the cloned methods. Figure 4.4 and Figure 4.12 present two examples of these two types of inconsistent program styles, i.e. different ways to validate the input parameters and handle exceptions. Investigating the cloned methods that have such inconsistent programming styles suggests that after we reconcile inconsistencies among these cloned methods, these cloned methods could also be refactored, for example, by extracting validity checking of input parameters into a utility method.

4.5.2.3 Refactoring clones using Java generics

Java generics support developing common data structures and algorithms that differ only in the set of types on which they operate.

To identify candidate clones that can be replaced with Java generic methods or classes, we formulate the following two queries based on the two characteristics of JavaIO library:

1. JavaIO supports reading and writing data of different primitive data types (e.g., short, char, int, long, double). Thus, we formulate a query to identify cloned methods that have *only differential typecasting statements*;
2. JavaIO supports reading and writing both byte (8-bit) data and char (16-bit) data. Thus, we formulate a query to identify cloned methods that have *only differential field-access and method-invocation statements*.

```
     Bits.getChar(byte[] b, int off)
26.  return (char)((b[off+1]&0xFF)<<0) +
27.                          (b[off+1]&0xFF)<<8));
     Bits.getShort(byte[] b, int off)
31.  return (short)((b[off+1]&0xFF)<<0) +
32.                          (b[off+1]&0xFF)<<8));
```

Figure 4.15. Differential typecast statements

It is surprising that the first query returns only one pair of cloned methods *Bits.getChar()* and *Bits.getShort()* as shown in Figure 4.15. Our inspection of JavaIO library reveals that this is because JavaIO mainly relies on bitwise shift and logic operations instead of explicit typecasting for processing data of different primitive data types.

The second query returns 26 pairs of cloned methods, including the cloned methods *PipedOutputStream.write(int)* and *PipedWriter.write(int)* listed in Figure 4.3. Although the two methods are textually identical, they actually have three instances of differential statements. Such differential statements also exist in other cloned methods returned by our query, such as methods *connect()*, *flush()*, *close()* of *PipedOutputStream* and *PipedWriter*. These differential statements reveal that the overall data and control flows are similar in many methods of two types of output classes (*PipedOutputStream* versus *PipedReader*), but the specific data operations are different.

In fact, these differential statements are resulted from parallel inheritance hierarchies in Java IO for processing byte data (input/output streams) and char data (readers/writers) respectively. JavaIO initially supported only byte data. To support char data, a separate hierarchy of classes was subsequently developed. The two parallel hierarchies share many similar data structures and processing steps. They may be restructured into one hierarchy using Java generic classes.

4.5.3 Refactoring Eclipse JDT-model unit tests

Next, we discuss our study on clone-based refactoring of JDT-model unit tests. Unit tests typically contain groups of test methods that form variations for a common testing purpose and therefore are similar to each other. In this study, we are interested in identifying clones that can be refactored using seed values, state machine, or assume/assert invariants testing patterns [198], thus reducing code duplication among test methods and improving test-case reuse.

4.5.3.1 Refactoring clones using seed values

A traditional unit test method tests a unit with fixed input value. It is necessary to develop several tests with variant input values to achieve a good coverage of the unit under test. These tests are often similar but also differ in the input values that are actually used.

We would like to refactor such duplicated unit tests into parameterized unit tests, using *seed-values* [198] (a pattern for parameterized unit testing) to provide concrete input values. To that end, we formulate a query searching for cloned test methods that have *only differential-operand and/or differential-constant-value statements*.

```
JavaSearchTests.testEnum06()
3672.   method = getMethod("setRole", new String[] {"Z"});
3673.   search(method, REFERENCES, ...);
JavaSearchTests.testVarargs03()
3702.   method = getMethod("vargs", new String[] {"QSt"});
3703.   search(method, ALL_REFERENCES, ...);
```

Figure 4.16. Seed values

Our query returns 173 pairs of cloned test methods in Eclipse JDT-model unit tests. Figure 4.16 presents one of them. The two methods test Java search API with different search entities (*setRole* versus *vargs*, and *Z* versus *QSt*) and search options (*REFERENCES* versus *ALL_REFERENCES*).

Investigating these 173 cloned test methods returned by our query suggests that cloned test methods for testing searching and formatting features usually have differential-operand and/or differential-constant-value statements. Such cloned test methods may be parameterized, using their differential operands and constant values as seed values, so that parameterized unit tests can verify the unit under test for a set of input values.

4.5.3.2 Refactoring clones using state machine

Eclipse JDT-model provides APIs for programmatically rewriting Java programs, such as creating a member (e.g. field or method) in a class. The corresponding unit tests for these APIs often share similar control and data flows but also differ in the program-rewriting APIs under tests.

We would like to refactor such cloned test methods into parameterized unit tests to enforce the flow of testing logics, using *state machine* [198] (another pattern for parameterized unit testing) to encapsulate program-rewriting APIs under test. To that end, we formulate a query searching for cloned methods that have differential method-invocation statements and/or differential blocks of program-rewriting APIs.

	ASTRewritingStatementsTest.testSwitchStatement7()
3956.	ListRewrite listRewrite = rewrite.getListRewrite(...)
3957.	listRewrite.**replace**(assignment, switchCase, null);
3959.	String preview = evaluateRewrite(cu, rewrite);
	ASTRewritingStatementsTest.testSwitchStatement9()
4098.	ListRewrite listRewrite = rewrite.getListRewrite(...)
4099.	listRewrite.**remove**(assignment, null);
4100.	listRewrite.**insertAfter**(switchCase,assignment,null):
4102.	String preview = evaluateRewrite(cu, rewrite);

Figure 4.17. State machine

Our query returns 153 pairs of cloned test methods. Figure 4.9 presents an example of these cloned methods that tests different ways to create a class member (field versus method). Figure 4.17 presents another example that tests different ways (*replace* versus. *remove* then *insertAfter*) to rewrite switch statements in an AST.

Investigating these 153 cloned test methods reveals that cloned test methods for testing program-rewriting APIs often invoke different program-rewriting APIs (e.g., *createField*, *createMethod*, *remove*, *insert*, *copy*, *move*, *replace*), or invoke some program-rewriting APIs in different order. The invocations of these program-rewriting APIs can be encapsulated into state machines [198] that programmatically rewrite Java programs. Then, given state machines of a set of program rewriting APIs and a parameterized unit test, one can use testing framework, such as Pex [198], to instantiate sequences of state transitions for testing the relevant program-rewriting APIs.

4.5.3.3 Refactoring clones using assume/assert invariants

JDT-model unit tests often assert similar set of properties that a unit under test should hold before and after exercising the unit under test, for example whether the parent AST node is not null or the class contains a specific member.

We would like to extract these similar assertions before and after exercising the unit under test into assume/assert invariants. To that end, we formulate a query searching for cloned test methods that have *at least two matched method-invocations of assertxxx() methods*, as Eclipse JDT-model unit tests name assertion methods in form of *assertxxx()*.

Our query returns 137 pairs of cloned test methods. Figure 4.18 presents one example. The two methods test the APIs of two different types of AST nodes, array creation versus switch statement. However, they share the same set of assertions about the AST under test, i.e. *modificationCount* > *prevCount*, *x.getAST()==this.ast*, and *x.getParent()==null*.

```
ASTTest.testArrayCreation()
7994.  final ArrayCreation x = this.ast.newArrayCreation();
7995.  assertTrue(this.ast.modificationCount > previousCount);
7997.  assertTrue(x.getAST() == this.ast);
7998.  assertTrue(x.getParent ()== null);
ASTTest.testSwitchStatement()
5891.  final SwitchStatemnt x = this.ast.newSwitchStatement();
5892.  assertTrue(this.ast.modificationCount > previousCount);
5894.  assertTrue(x.getAST() == this.ast);
5895.  assertTrue(x.getParent() == null);
```

Figure 4.18. Assume invariant

Investigating these 137 cloned test methods reveals that cloned test methods for JDT AST/DOM APIs often contain similar set of assertions before and/or after exercising the AST/DOM API under test. These similar assertions may be extracted as assume and/or assert invariants. These invariants can not only remove duplicate assertions across test methods, but also better ensure consistent verifications of test assumptions and results.

4.6 Related Work

Researchers have presented many techniques to detect code clones [9,12,20,55,80,104]. Roy and Cordy [144] and Koschke [101] provide comprehensive surveys of existing clone detection techniques. Clone detectors typically report large number of clones in industrial systems, while it is common that only small number of them is actually useful for specific maintenance tasks, such as refactorings.

The effectiveness of clone detection techniques has usually been evaluated in terms of precision and recall metrics of the detected clones, such as in the quantitative evaluation of clone detection techniques reported in Roy and Cordy [145] and Bellon et al. [21]. However, the precision and recall metrics do not indicate the usefulness of the detected clones for a specific maintenance task.

Researchers proposed clone analysis approaches to aiding the interpretation and management of software clones. For example, Genimi [165] uses a scatter plot to visualize code clones detected by CCFinder [80] and also computes several code metrics of clones

to aid clone analysis. Balazinska et al. [9, 10] define a clone classification based on the differences between the token sequences forming the clones. This clone classification helps to measure the reengineering opportunities of clones. CP-Miner [109] finds bugs based on inconsistent identifiers between clones. One major limitation of these clone analysis approaches is that they examine only the information of clones, ignoring contextual information in which clones occur.

Other approaches perform simple syntactic analysis of clones to aid the understanding of clones. For example, Kapser and Godfrey [85] classify code clones through the syntactic analysis of locality of clones. Jiang et al. [77] consider the inner most syntactic constructs that enclose clones as contexts and identify three types of contextual inconsistencies in clones in an ad-hoc manner. In contrast, our CloneDifferentiator raises the contextual analysis of code clones to PDGs that capture much more contextual information than existing work. Furthermore, Clone-Differentiator exploits efficient graph differencing algorithm to systematically detect contextual differences of clones.

Query-based approaches have been proposed for supporting program understanding. The underlying idea of these approaches is that software tools can automatically compute elementary information, and then developers can combine and query the elementary information in a task-oriented manner to assist a specific maintenance task at hand. For example, Xing and Stroulia [172] proposed to detect change patterns in software evolution by querying the elementary design changes reported by UMLDiff. Zhang et al. [184] present the CloneAnalyzer tool that support query-based filtering of code clones. CloneAnalyzer does not support contextual analysis and differencing of clones as CloneDifferentiator does.

CloneDifferentiator is a new application of GenericDiff for the purpose of comparing the PDGs of clones. CloneDifferentiator performs automatic contextual analysis of code

clones based on the PDG differencing results by GenericDiff. We present our CloneDifferentiator tool in [173], which focuses on the implementation challenges and the visualization features of the tool support. In contrast, this chapter describes the fundamental concepts of our CloneDifferentiator approach, discusses in detail contextual differences of clones and their implications on understanding clones, and reports two empirical studies for evaluating our approach.

4.7 Threats to Validity

Our CloneDifferentiator tool currently uses CloneMiner [12] to detect cloned methods. The surveys [101,144] on clone detection tools suggest that clones reported by different techniques may vary due to the diverse nature of detection techniques and similarity metrics. The different model differencing or code detection tool may affect the accuracy of our results. However, the choice of one proper technique from the similar candidates (such as CloneMiner for clone detection, GenericDiff for model differencing) should not have fundamental impact on our approach. The reason is that the techniques we chose are modern, advanced and comparable to the others in their corresponding domain. To see the solid proof for our assumption, further studies are required to evaluate our approach with respect to different clone detection, or the different model differencing techniques.

CloneDifferentiator now compares intra-method PDGs of cloned methods. It does not consider inter-method PDGs because it assumes that two different methods being invoked in cloned methods would perform different computation. This assumption holds in most cases and allows efficient contextual analysis of clones. CloneDifferentiator can be easily adapted to analyze inter-procedure PDGs around cloned methods, because inter-procedure PDGs are available in Wala, and GenericDiff can be easily configured to compare inter-method PDGs.

In this chapter, we showed that contextual differences of clones are useful for distilling useful clones for refactorings aiming at reducing code duplication. Cloning information has also been used for other types of software maintenance tasks, such as bug detection [77,79,103]. Further studies are required to investigate the usefulness and effectiveness of Clone-Differentiator for other types of maintenance tasks.

4.8 Summary

Similar yet still different features may manifest as software clones in a single product or across several products. Understanding the differences among these clones provides one way of understanding implementations of variant features. Furthermore, programmers cannot judge code clones without understanding differences. To perform maintenance tasks on clones, developers must examine the differences of clones induced by the program context in which clones occur.

In this chapter, we proposed an automated approach to identify contextual differences of code clones by applying graph differencing algorithm to the PDGs of clones' program contexts. We have implemented our approach in a tool called CloneDifferentiator. Our evaluation demonstrated that CloneDifferentiator can effectively distill clones that are valid candidates for various refactoring tasks aiming at reducing code duplication. It helps to reduce effort of post-detection analysis of clones in a task oriented manner.

In the future, we plan to conduct more empirical studies to enrich our taxonomy of contextual differences of clones. We believe this can open new opportunities to refine existing clone definitions from a new perspective (i.e., how clones are different). This can enhance the usefulness of cloning information in various software maintenance tasks.

5 Locating Features in Product Variants

This chapter is based on [179], which focus on the last but not the least step RQ3 introduced in Section 1.1 for variability analysis in domain engineering. We present the problem of feature location in product family. We reviewed the existing feature location approach for a single product, and proposed our model-differencing based approach to identifying and locating features among product variants.

5.1 Introduction

Once the RQ1 and RQ2 are resolved, the differences among the various product variants are distinguished respectively in requirements and in implementation. But still, it is not known that which variant feature' inclusion/exclusion enables/disables the corresponding code portion or method in the implementation. For example, suppose we represent the features supported by a product as set f, then for $WFMS^{Fudan}$ and $WFMS^{Shanghai}$, we have f_{Fudan} and $f_{Shanghai}$ respectively. Suppose we represent the code implementation as c, and then we also have c_{Fudan} and $c_{Shanghai}$. After RQ1 and RQ2 are resolved, the information at requirement level about $f_{Fudan} - f_{Shanghai}$ and $f_{Shangha} - f_{Fudan}$ is known. And it is also available that the information at implementation level about $c_{Fudan} - c_{Shanghai}$ and $c_{Shangha} - c_{Fudan}$. But for a feature like **DelegationLock** in the set $f_{Fudan} - f_{Shanghai}$, it is still interesting to know the counterpart of feature-relevant code in the $c_{Fudan} - c_{Shanghai}$. Actually, the problem is called as "feature location" in the product variants.

Ideally, by applying the information retrieval technique such as Latent Semantic Indexing (LSI), the method **FeeInfo.initInfo()** is retrieved respectively in PFM^{Fudan} and $PFM^{Shanghai}$ for feature **DelegationLock** and **OperationLock** according to their own feature descriptions. By combing the knowledge of the domain analysis at requirement level (see Figure 3.2) and the clone analysis at implementation level (see Figure 4.1), we can further

infer that feature **DelegationLock** in PFM^{Fudan} and feature **OperationLock** in $PFM^{Shanghai}$ are two different features. And comparing these two cloned methods reveals that these two features **DelegationLock** and **OperationLock** have contextual differences represented by PDG, which actually is due to the variability.

The above example is a simple illustration of the key idea of our approach to supporting effective feature location in a software product family to help domain analysis. However, situations are more complicated in real-world software product families. First, features and their implementations may evolve from one product variant to another. As a result, same features and code units may appear to be different in different product variants. *How can we precisely determine correspondences between features (or code units) across product variants?* Second, product variants share common features and their implementations. *How can we systematically partition product variants so that each partition is disjoint and consists of a minimal subset of features (or code units)?*

Besides, the recent large-scale empirical study [168] showed that IR-based approaches to feature location do not perform well on software data, even for a single software product. Direct application of these approaches to a software product family will be even worse due to significant increase of search space. Furthermore, same features and their implementations may vary across product variants due to evolution and customization of different product variants. This makes it difficult to first apply IR techniques for feature location in each product variant and then merge feature location results.

In this chapter, we present a systematic approach to supporting Feature Location in Software Product Family (FL-SPF for short) that consists of a set of product variants supporting overlapping but also different sets of features. Our goal is to *identify common code units that implement overlapping features across product variants*. Our approach effectively incorporates *software differencing*, *Formal Concept Analysis (FCA)*, and *IR* techniques.

Chapter 5 Locating Features in Product Variants

Software differencing helps to identify distinct features (or code units) in a software product family, which represent corresponding features (or code units) across product variants. FCA then groups distinct features (or code units) into disjoint and minimal partitions by analyzing commonality and differences of product variants. These two steps of analysis help to reduce search space for feature location in software product family. Finally, given a feature partition and the corresponding code-unit partition, Latent Sematic Indexing [41] (LSI) is used to identify code units that implement a specific feature.

We have implemented our approach and conducted evaluation with nine product variants of Linux kernel. These nine Linux kernel product variants consist of 3146 features and about 342K code units (functions and data structures in our evaluation). Our evaluation shows that:

- Software differencing can significantly reduce search space for feature location in software product family by determining distinct features (or code units) in software product family. In our evaluation we only need to deal with 966 distinct features and about 100K distinct code units in subsequent analysis;
- FCA can effectively group features (or code units) into disjoint and minimal partitions. In our evaluation, the resulting partitions consist of only 8.9±14 (average± standard deviation) features and 948.8±1540 code units;
- Our approach always outperforms a direct application of IR technique in the subject product family: compared with the best performance of direct application of LSI, the worst performance of our approach still identifies relevant program elements for 25.7% more features, achieves 32% higher mean average precision, and requires investigating 8% less code units before encountering the first relevant code unit.

5.2 Related Work

Identifying code units (e.g. functions) that implement a specific feature in a software system is known as feature location [24,139]. A feature in a software system represents a distinct aspect of the system that is accessible to developers and users [81]. Software maintenance involves adding new features to a system, improving and reengineering existing features, and removing unwanted features (e.g. bugs). Feature location is one of the most important and common activities performed by developers during software maintenance [44], because no maintenance task can be completed without first locating and understanding the code that is relevant to the task at hand.

A comprehensive survey of feature location techniques can be found in [44]. Feature location techniques mainly employ textual, static, and dynamic analysis. Textual approaches [118,119,134,135,148] analyze words in source code using IR techniques. Static analysis [62,102,164] examines structural information such as program convergence, control and data dependencies. Dynamic analysis [49,169,170] examines execution traces of feature-specific execution scenarios. Hybrid approaches [47,68,135,136,185] combine two or more types of analysis with the goal of using one type of analysis to compensate for the limitations of another, thus achieving better results.

Our recent empirical study [168] investigates the effectiveness of 10 IR techniques on a very large Linux kernel dataset. This study suggests that one must exploit unique characteristics of software data so that IR techniques may perform similarly well on software data, compared with on natural language articles that IR techniques are designed for. Inspired by this study, we proposed several new techniques for feature location. For example, based on the fact that features are not independent of each other, we proposed to combine IR techniques with graph matching to solve feature location as an iterative context-aware graph matching problem [130]. In this work, we propose to analyze commonalities and

differences of product variants in a software product family by software differencing and FCA techniques so that IR technique can achieve better results for feature location in software product family.

Software differencing plays a fundamental role in many software maintenance tasks, including design evolution and variability analysis. For example, Xing and Stroulia presented UMLDiff algorithm [172] for analyzing design evolution of object-oriented software. In our recent work [180], we compared product feature models to understand how product variants in a software product family evolve at requirement level. Duley et al. [45] proposed to use lexical similarity and bipartite graph matching for differencing hardware programs. Such differencing techniques can help to identify distinct features (or code units) in software product family, and thus reduce search space for IR-based feature location.

FCA has been used to aid feature location analysis. For example, to alleviate the difficulty to formulate distinct feature-specific execution scenarios, Eisenbarth et al. [48] applied FCA to isolate features through analysis of overlapping scenarios. In fact, this inspired our feature location approach in software product family. We consider product variants as "scenarios" that support similar but also different sets of features. Thus, FCA can be applied to group features into disjoint, minimal partitions through analysis of commonality and differences of product variants. FCA has also been used for post-mortem analysis of feature location results. For example, Poshyvanyk and Marcus [134] use FCA to group IR-based feature location results according to common topics. Their goal is to reduce the interpretation effort of IR search results by providing additional structure on top of search results. In contrast, we use FCA to pre-processing input search space to which IR techniques apply.

5.3 The Approach

In this section, we describe input data for our approach. We also discuss how to identify distinct features (or code units) in software product family by software differencing, how to group features (or code units) into disjoint, minimal partitions by FCA, and how to apply LSI for feature location.

5.3.1 A running example

Consider a software product family of five document viewers and editors shown in Table 5.1. The product *Viewer1.0* ($V_{1.0}$) is a simple text viewer that supports basic features such as file open and text viewer. *Viewer2.0* ($V_{2.0}$) is an advanced text viewer that supports not only basic features but also two more features (*Find* and *Copy to Clipboard*). *Editor1.0* is a simple editor that supports basic texting editing features such as *Edit*, *Undo/Redo* and *Save*. *Editor1.1* ($E_{1.1}$) does not support instable feature *Undo/Redo*; it ports feature *Find* from *Viewer2.0* and extends it into an enhanced feature *Find/Replace*. *Editor2.0* ($E_{2.0}$) supports all basic texting editing features; it ports feature *Copy* from *Viewer2.0* and extends it into an enhanced feature *Copy/Cut*; it also supports a new feature *Paste*.

Table 5.1. Feature sets of document viewers/editors

Products	Features
Viewer1.0	Base (Text Viewer, Open)
Viewer2.0	Base, Find, Copy
Editor1.0	Base, Edit, Save, Undo/Redo
Editor1.1	Base, Edit, Save, Find/Replace
Editor2.0	Base, Edit, Save, Undo/Redo, Copy/Cut, Paste

5.3.2 Input data

For a product variant in a software product family, our FL-SPF approach takes as input a set of features that the product variant supports, and a static program model built from the implementation of the product variant.

Each feature of a product variant is identified by a *name* and is described using some natural language *description*. Such feature information can be extracted from release notes, user manuals, or feature models of the software system. For example, Figure 5.1 shows a feature of Linux Kernel, expressed in Kconfig (feature modeling language of Linux Kernel) [151]. It describes a feature that supports microcode patch loading for Intel processors (see Section 5.4.2).

```
config CONFIG_MICROCODE_INTEL
        bool "Intel microcode patch loading support"
        depends on CONFIG_MICROCODE;   select FW_LOADER
        default y
---help---
        This option enables microcode patch loading support for Intel
      processors …
```

Figure 5.1. A feature in Linux kernel

The static program model of product implementation is a graph. The node set contains code units of interest for feature location such as functions and data structures. Each code unit is associated with a set of properties, such as identifier and comments. The edge set contains relations between code units, such as function call and data-structure usage. Such program models can be reverse-engineered from product implementation and have been widely used for program understanding and maintenance.

5.3.3 Identifying distinct features (or code units) in Software Product Family by software differencing

Due to evolution and customization of software products, product variants in a software product family may contain different versions or variants of the same features (or code units). FL-SPF uses software differencing techniques (for example [45] and [172]) to determine correspondences of features (or code units) among product variants. Analyzing feature (or code-unit) correspondences allows FL-SPF to determine distinct features (or

code units) in the software product family. These distinct features (or code units) represent a set of corresponding features (or code units) across several product variants. They significantly reduce search space for feature location in software product family, because FL-SPF no longer needs to deal with features and code units of each individual product variant.

5.3.3.1 Differencing feature sets of product variants

At feature level, FL-SPF examines lexical similarities of feature descriptions and uses bipartite matching to determine correspondences of features of two product variants.

Given feature sets of two product variants F_1 and F_2, FL-SPF first identifies a subset of same-name features F_{sn} between the two sets F_1 and F_2. For those features that do not have same-name counterparts, i.e. $f_1 \in F_1 \backslash F_{sn}$ and $f_2 \in F_2 \backslash F_{sn}$ and $f_1.name \neq f_2.name$, FL-ESPF computes their lexical similarities pair-wisely using Longest Common Subsequence (LCS) of their feature descriptions. LCS considers ordering of words, which is important for names of program elements. While for information retrieval task, bags of word is more general. FL-ESPF then builds a bipartite graph (N_1, N_2, E): N_1 and N_2 are disjoint node sets and they represent features $f_1 \in F_1 \backslash F_{sn}$ and $f_2 \in F_2 \backslash F_{sn}$ respectively, and weighted edges in E represent lexical similarities between features $f_1 \in F_1 \backslash F_{sn}$ and $f_2 \in F_2 \backslash F_{sn}$. FL-SPF selects an optimal matching between features $f_1 \in F_1 \backslash F_{sn}$ and $f_2 \in F_2 \backslash F_{sn}$ using state marriage algorithm [56].

FL-SPF reports differencing results of feature sets F_1 and F_2 as three sets of features: a set of matched features $F_1 \hat{} F_2$ (i.e. pairs of matched features, one from each product variant), and two sets of unmatched feature $F_1 \backslash F_2$ and $F_2 \backslash F_1$ (i.e. unmatched features that exist only in F_1 but not in F_2 or in F_2 but not in F_1).

For example, consider *Viewer2.0* ($V_{2.0}$) and *Editor1.1* ($E_{1.1}$) in Table 5.1. The two product variants have only one pair of same-name feature, i.e. $V_{2.0}.Base$ and $E_{1.1}.Base$. FL-SPF further identifies the correspondence between feature $V_{2.0}.Find$ and feature

$E_{1.1}$.*Find/Replace* based on lexical similarity of their descriptions and bipartite matching. Thus, FL-SPF reports that $F_{V2.0}$^$F_{E1.1}$={$V_{2.0}$.*Base_*$E_{1.1}$.*Base*, $V_{2.0}$.*Find_*$E_{1.1}$.*Find/Replace*}, $F_{V2.0}$\$F_{E1.1}$={$V_{2.0}$.*Copy*}, and $F_{E1.1}$\$F_{V2.0}$={$E_{1.1}$.*Edit*, $E_{1.1}$.*Save*}.

5.3.3.2 Differencing program models of product variants

At implementation level, FL-SPF uses a variant of UMLDiff algorithm [171] to compare program models of two product variants. Given two program models P_1 and P_2, UMLDiff determines correspondences of code units based on lexical similarity of their identifiers and comments and structural similarity of their relationships with other elements.

Based on UMLDiff's differencing report, FL-SPF reports differencing results of program models of two product variants as three sets of code units: a set of matched code units P_1^P_2 (i.e. pairs of matched code units, one from each product variant), and two sets of unmatched code units P_1\$P_2$ and P_2\$P_1$ (i.e. unmatched code units that exist only in P_1 but not in P_2 or in P_2 but not in P_1).

5.3.3.3 Identifying distinct features (or code units)

The correspondence relations as defined by matched features (or code units) across product variants partition the union of feature sets (code-unit sets) of all product variants into a set of equivalence classes. Each equivalence class consists of a set of corresponding features (or code units) of several product variants; it represents a distinct feature (or code unit) in the software product family.

For example, the software product family in our running example has five product variants that have in total 18 features. These 18 features of individual product variants can be represented as seven distinct features in the software product family, including <u>*Base*</u>={$V_{1.0}$.*Base*, $V_{2.0}$.*Base*, $E_{1.0}$.*Base*, $E_{1.1}$.*Base*, $E_{2.0}$.*Base*}, <u>*Find/Replace*</u>={$V_{2.0}$.*Find*, $E_{1.1}$.*Find/Replace*}, <u>*Copy/Cut*</u>= {$V_{2.0}$.*Copy*, $E_{2.0}$.*Copy/Cut*}, <u>*Edit*</u>={$E_{1.0}$.*Edit*, $E_{1.1}$.*Edit*,

$E_{2.0}$.*Edit*}, $\underline{Save}$={$E_{1.0}$.*Save*, $E_{1.1}$.*Save*, $E_{2.0}$.*Save*}, $\underline{Undo/Redo}$={$E_{1.0}$.*Undo/Redo*, $E_{2.0}$.*Undo/Redo*}, $\underline{Paste}$= {$E_{2.0}$.*Paste*}.

5.3.4 Grouping features (or code units) into disjoint, minimal partitions by FCA

Product variants in a software product family support overlapping but also different sets of features. We use FCA technique to group features (or code units) in the software product family into disjoint, minimal partitions through combination of commonalities and differences of product variants. Such feature and code-unit partitions further reduce space for feature location in software product family, because FL-SPF no longer needs to perform feature location on the entire set of features and code units.

5.3.4.1 Basic concepts of FCA

FCA deals with a formal context (I, O, F) that defines a relation $I \subseteq O \times F$ between a set of objects O and a set of attributes F. A tuple $c=(O_i, F_j)$ is a concept of a given formal context (I, O, F) iff $O_i \subseteq O$, $F_j \subseteq F$ and every object $o \in O_i$ has every attribute $f \in F_j$. That is, the concept c identifies a set of objects that share a set of common attribute. The set of objects O_i is called the extent of the concept c, and the set of attributes F_j is called the intent of c. The set of all concepts of a given formal context forms a partial order $\leq$: $c_1=(O_1,F_1) \leq c_2=(O_2,F_2)$ iff $O_1 \subseteq O_2$ or equivalently $F_1 \supseteq F_2$. This partial order induces a concept lattice. Intuitively, the higher a concept is in the concept lattice, the more general its extent is (i.e. more objects) but the more specific its intent is (i.e. less common attributes).

5.3.4.2 Determining feature-level partitions by FCA

In our application of FCA, we define a formal context as follows. We consider product scenarios as objects in a formal context, and consider distinct features of the software product family as attributes. A relation between a product scenario and a distinct feature defines that the distinct feature can be found in the product scenario.

Such a formal context can be constructed based on differencing results of feature sets of product variants in the software product family. For any two product variants P_1 and P_2, we construct three product scenarios $P_1 \wedge P_2$, $P_1 \setminus P_2$, and $P_2 \setminus P_1$. A distinct feature can be found in the product scenario $P_1 \wedge P_2$ (or $P_1 \setminus P_2$, $P_2 \setminus P_1$) iff the feature of product variants represented by the distinct feature is in the matched feature set $F_1 \wedge F_2$ of two product variants (or unmatched sets F_1/F_2, F_2/F_1). Consider *Viewer2.0* ($V_{2.0}$) and *Editor1.1* ($E_{1.1}$) as an example. Based on differencing results of feature sets of these two product variants, three product scenarios can be constructed as follows: $V_{2.0} \wedge E_{1.1}$={*Base*, *Find/Replace*}, $V_{2.0} \setminus E_{1.1}$={*Copy/Cut*}, and $E_{1.1} \setminus V_{2.0}$={*Edit*, *Save*}.

This formal context summarizes commonalities and differences among product variants. Given such a formal context, we use ConceptExplorer [192] to induce a concept lattice. Each concept in the lattice represents a partition of search space of the software product family; each partition shows how we can isolate certain features and their implementations through combination of product scenarios.

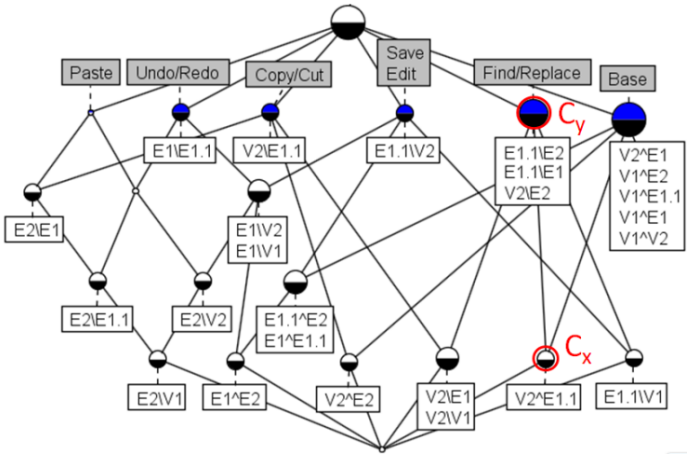

Figure 5.2. The concept lattice of document viewers/editors

Figure 5.2 presents the resulting concept lattice of our running example. This lattice has five levels and 20 concepts (i.e. nodes). The node associated with a set of features (grey box) represents the most general concept that has those features as its intent; while the

node associated with a set of product scenarios (white box) represents the most special concept that has those product scenarios as its extent. The intent and extent of a concept can be derived as follows: the extent of a concept is the union of all product scenarios at and below the concept node; the intent of the concept is the union of all features at and above the concept node. For example, the intent of concept C_x is feature set {*Base*, *Find/Replace*}, and the extent of C_x is product scenario set $\{V_{2.0}\text{^}E_{1.1}\}$. As another example, the intent of concept C_y is {*Find/Replace*}, and the extent of C_y is $\{V_{2.0}\text{^}E_{1.1}, E_{1.1}\backslash V_{1.0}, V_{2.0}\backslash E_{2.0}, E_{1.1}\backslash E_{1.0}, E_{1.1}\backslash E_{2.0}\}$.

In such a concept lattice, we are interested in concept nodes associated with a set of features (for example the concept C_y). These concepts represent the most general set of product scenarios that share the least common set of features, i.e. the finest-grained partitions of search space of the software product family. These concepts show how we can obtain a minimal subset of features and code units for feature location in the software product family.

Because we examines product scenarios (defining not only commonalities but also differences of product variants), we are able to produce more fine-grained partitions of search space of a software product family than examining only commonalities of product variants. For example, examining only commonalities between *Viewer2.0* and *Editor1.1*, we cannot separate feature *Base* from feature *Find/Replace*. However, by considering differences between several relevant product variants (i.e. features in $V_{2.0}$ but not in $E_{2.0}$, and features in $E_{1.1}$ but in neither $V_{1.0}$, $E_{1.0}$, nor $E_{2.0}$), we can then perfectly isolate feature *Find/Replace*. This is how we end up with the concept C_y in the lattice in Figure 5.2.

5.3.4.3 Building code-unit level partitions

Given a feature-level partition (i.e. a concept computed by FCA), FL-SPF first analyzes the relevant product scenarios to determine program models it has to compare. For exam-

ple, given the concept C_y, FL-SPF will compare program models of *Viewer2.0* and *Editor1.1*, *Viewer2.0* and *Editor2.0*, *Editor1.1* and *Viewer1.0*, *Editor1.1* and *Editor1.0*, and *Editor1.1* and *Editor2.0.* FL-SPF then determines distinct code units based on differencing results of program models (see Section 5.3.3), and build the corresponding code-unit level partition for locating implementations of the distinct features of the given feature-level partition. For the concept C_y, its corresponding code-unit level partition will include distinct code units that are in both *Viewer2.0* and *Editor1.1* but in neither *Viewer1.0*, *Editor1.0* nor *Editor2.0*. This code-unit level partition will be used for locating implementations of feature {*Find/Replace*}.

5.3.5 Feature location by LSI

Given a set of distinct features *F* and a set of distinct code units *CU*, FL-SPF uses LSI technique to identify code units in *CU* implementing features in *F*. Our application of LSI for feature location is similar to existing studies [119,168]. It involves extraction of bag-of-word representation for features and code units, and retrieval of code units implementing a feature.

5.3.5.1 Building feature queries

As discussed in Section 5.3.3, a distinct feature represents a set of corresponding features across several product variants. The name and description of these corresponding features may be different due to evolution and customization of product variants. Thus, given a distinct feature, FL-SPF needs to first merge name and description of these corresponding features to generate name and description of the distinct feature. FL-SPF adopts a merging strategy similar to that of CVS for merging textual documents. However, FL-SPF does not attempt to resolve conflicts in feature descriptions. Instead, it simply retains all conflicting fragments of feature descriptions.

Then, FL-SPF converts each distinct feature into a bag of words as follows. It first extracts word tokens from the name and description of the distinct feature. It uses white space and underscore as delimiters. Then, it removes commonly used stop words of little meaning, e.g. is, are, will, have, etc. Finally, it reduce every word token to its root form, for example swap, swapping, swapped are reduced to the same root form swap. The bag-of-words representation of distinct features will be used as feature queries to an IR engine.

5.3.5.2 Building corpus of code units

Similar to the processing of distinct features, given a distinct code unit that represents a set of corresponding code units across several product variants, FL-SPF first generates code fragments of the distinct code unit by merging code fragments of the corresponding code units. It uses the same merging strategy as that for merging feature descriptions.

Then, FL-SPF converts each distinct code unit into a bag of words as follows. It first extracts word tokens involving string literals, variable names, function names, parameter names, and words in the source code comments. Then, it removes word tokens corresponding to language keywords, e.g. if, else, while, for, etc. Finally, it also reduces every word token to its root form. The bag-of-words representation of distinct code units constitutes a corpus of code-unit documents for feature location.

5.3.5.3 Code-units retrieval using LSI

We feed bags-of-words of features and code units to an LSI engine for feature location. We use SemanticVectors [200] for LSI analysis. LSI engine first analyzes all bags of words to build a topic model. Then, it considers a feature query and every code-unit document as a vector of weights; each weight is the likelihood for the query or a code unit belonging to a particular topic. It measures the similarities between the feature query and code-unit documents using cosine similarity of feature-query vector and code-unit vectors.

It finally returns a list of code units ordered based on their similarity scores against the feature query.

5.4 Linux Kernel Dataset

In this section, we summarize the Linux kernel dataset used for the evaluation of our approach. We describe how we extract from this dataset: 1) features with textual descriptions that can be used as queries for feature location; 2) code units that form corpus for feature location; and 3) ground truth links between features and their implementing code units for evaluating the effectiveness of our feature location approach.

5.4.1 Dataset

The Linux kernel [195] is an open source operating system kernel. It was originally developed for 32-bit x86-based PCs, but has since been evolved into a software product line [151], which consists of thousands of features that can be configured in order to generate specific kernel products for a vast combinations of architectures, subsystems, device drivers, etc.

Table 5.2. Nine product variants of Linux kernel

Product Variants	#Features	#Code Units
2.6.27.9 def	306	34060
2.6.27.9 r1	403	44275
2.6.27.9 r2	412	43913
2.6.31.9 def	348	39565
2.6.31.9 r1	458	55585
2.6.31.9 r2	325	29936
2.6.37 def	402	45312
2.6.37 r1	164	17622
2.6.37 r2	328	32461
Total	**3146**	**342729**

In this evaluation, we used nine product variants derived from three stable releases of the Linux kernel 2.6.27, 2.6.31, 2.6.37 released on December 14th, 2008, December 18th,

2009, and January 5th, 2011, respectively. These product variants were built for a Dell Precision M6400 workstation based on default and two random configurations shipped with each kernel release.

We extract a dataset that contains 3146 features and 342729 code units (functions and data structures). These nine product variants comprise 350±86 (average±standard deviation) features and 38081±10988 code units. Our statistics is consistent with earlier well-known study on Linux kernel evolution [59]. That is, although the full source tree of Linux kernel is very large, a specific kernel product (built for a specific platform, devices, features, etc.) is likely to comprise only a small portion of the full source tree.

5.4.2 Extracting features sets

The Linux kernel manages features using a feature modeling language (Kconfig) and its accompanied configuration tool [151]. Each feature defined in Kconfig model has a configuration symbol, a short prompt, and a free-form textual description, together with other information such as datatype, default value of the feature, and dependencies between features.

Figure 5.1 lists a feature excerpt expressed in Kconfig. It defines a feature CONFIG_MICROCODE_INTEL (configuration symbol) that provides "Intel microcode patch loading support" (short prompt). Its feature description states "*This option enables microcode patch loading support for Intel processors. ...*". This feature is of Boolean type and it is included in kernel products by default. Furthermore, it depends on another feature whose configuration symbol is CONFIG_MICROCODE, and selecting CONFIG_MICROCODE_INTEL requires selecting feature FW_LOADER as well.

To build a specific kernel product, a user initializes and modifies a feature configuration based on the Kconfig model, using the accompanied Kconfig tool. In this evaluation, we built kernel products for a Dell Precision M6400 workstation based on default and two

random configurations shipped with each kernel release. The resulting kernel products constitute a software product family of nine product variants for evaluating the effectiveness of our approach.

We extended the Kconfig tool to extract the set of features that a specific kernel product supports. This tool extracts configuration symbol, short prompt and textual description of these features. The configuration symbol is treated as feature name, and short prompt and textual description is treated as feature description. Such sets of features of kernel products serve as feature-level input to our feature location approach.

5.4.3 Reverse-engineering program models

The feature configuration process by users results in a set of symbol-value pairs. These symbol-value pairs are used by Linux kernel in its makefiles and source code (as C preprocessors) to control which directories, files, and conditional blocks are enabled for compilation.

For example, if the Intel microcode feature is selected for a kernel product (i.e. CONFIG_MICROCODE_INTEL=y), functions and data structures defined in *microcode_intel.c* will be compiled. Then, the built kernel product will support microcode patch loading for Intel processors.

We developed the tool *progmodel* based on the C parser in the Sparse library [202] that has been integrated with Linux kernel build system. This allows *progmodel* to reverse-engineer static program model of the kernel product, as it is built according to a feature configuration. Note that reverse-engineered program model consists of only code units that implement features that a specific kernel product supports. Such program models serve as implementation-level input to our feature location approach.

5.4.4 Establishing ground truth

The Kconfig model, makefiles, and C preprocessors used by Linux kernel allow us to establish the ground truth links between features and their implementing code units.

For example, by analyzing the usage of the configuration symbol CONFIG_MICROCODE_INTEL in the makefiles and source code, we can establish that this feature has 17 implementing code units, including functions such as *collect_cpu_info()*, *request_microcode_fw(), init_intel_ microcode()*, and data structures such as *microcode_intel* and *extended_signature*.

We developed a tool to automatically extract such ground truth links. The availability of this ground truth allows us to systematically evaluate the effectiveness of our approach to feature location in software product family with thousands of features and hundreds of thousands of code units.

5.5 Results

We now review measures used in our evaluation and then present the results of the effectiveness of our approach to feature location in software product family.

5.5.1 Evaluation measures

We use three measures to evaluate the effectiveness of our approach: *Percentage of Relevant Queries* (*PRQ*), *Mean Average Precision* (*MAP*), and *Average Percentage of Code Units Investigated* (*APCUI*). These measures are commonly used to evaluate IR techniques [168].

1. struct extended_signature
2. struct extended_sigtable
3. request_microcode_fw(int,struct device*)
4. num_booting_cpus()
5. get_ucode_fw(void*,void*,unsigned int)
6. mtrr_cleanup(unsigned int)
7. mtrr_trim_uncached_memory(unsigned long)
8. _fsnotify_inode_delete(struct inode*)
9. init_intel_microcode()
10. get_ucode_user(void*,void*,unsigned int)

Figure 5.3. The top 10 returned code units for the Intel microcode feature

Percentage of Relevant Queries (PRQ). This measure quantifies the percentage of the feature queries that can return at least one relevant code unit in returned code units:

$$PRQ = \frac{\sum_{q=1}^{Q} B\left(\sum_{i=1}^{N_q} r(i)\right)}{Q}$$

where Q is the number of all feature queries, N_q is the length of the ordered list of code units returned by the query q, and *r(i)* is a binary function that returns 1 if the code unit at the position i is relevant to (i.e. in the ground truth of) the query q and return 0 otherwise, and *B(.)* is a Boolean function that return 1 if its parameter is not zero and 0 otherwise. Obviously, higher *PRQ* values mean better feature location results.

Figure 5.3 presents the top 10 (i.e N_q=10) functions and data structures returned for the Intel microcode feature. Six of the 10 returned code units ranked 1st, 2nd, 3rd, 5th, 9th, and 10th are relevant, i.e. $\forall i \in \{1,2,3,5,9,10\}$, r(i)=1. Thus, $\sum_{i=1}^{10} r(i) = 6$, and *B(.)* returns 1 for the query of the Intel microcode feature. After we obtained *B(.)* values for all feature queries, we can easily compute the *PRQ* value.

Mean Average Precision (MAP). MAP is the mean of average precision scores for a set of queries processed by an IR technique: $MAP = \frac{\sum_{q=1}^{Q} AP_q}{Q}$, where Q is the number of feature queries that return at least one relevant code unit, AP_q is the average precision score for the query q and defined as follows:

$$AP_q = \frac{\sum_{i=1}^{N_q}\left(P(i) \times r(i)\right)}{\sum_{i=1}^{N_q} r(i)}$$

where N_q and *r(i)* have the same meaning as above, and *P(i)* is the precision for the code unit at the position i and defined as follows: $P(i) = \frac{\sum_{j=1}^{i} r(i)}{i}$.

AP_q quantifies the precision of each code unit in the list based on the code unit's relevance and its position in the list, and then takes a weighted average of the precision of all code units in the list, giving more weights to relevant code units ranked higher.

For the query results of the Intel microcode feature, $\forall i \in \{1,2,3,5,9,10\}$ r(i)=1, then we have P(1)=1/1, P(2)=2/2, P(3)=3/3, P(4)=3/4, P(5)=4/5, P(6)=4/6, P(7)=4/7, P(8)=4/8, P(9)=5/9, P(10)=6/10. Thus, the average precision for this particular query result is $\frac{\frac{1}{1}+\frac{2}{2}+\frac{3}{3}+\frac{3}{4}\times 0+\frac{4}{5}+\frac{4}{6}\times 0+\frac{4}{7}\times 0+\frac{4}{8}\times 0+\frac{5}{9}+\frac{6}{10}}{6} = 0.826$.

After we obtain the average precision for all features, we can easily compute the MAP value. Obviously, higher MAP values mean better feature location results.

Average Percentage of Code Units Investigated (APCUI). This measure quantifies on average what percentages of the code units returned by a query need to be investigated before the first relevant document appears:

$$APCUI = \frac{\sum_{q=1}^{Q} \frac{I_q}{N_q}}{Q}$$

where Q is the number of feature queries that return at least one relevant code unit, and I_q is the smallest number such that $r(I_q)$=1^$\forall i \in \{1, \ldots, I_q - 1\}, r(i) = 0$. Obviously, smaller *APCUI* values mean better feature location results.

For the query of the Intel microcode feature, a relevant code unit appears at the position 1 in the list and thus $I_q=1$. Since the length of the list is 10 (i.e. N_q=10), the I_q/N_q value for this query is *1/10=0.1*. After we obtain I_q/N_q values for all feature queries, we can easily compute *APCUI* value.

5.5.2 Distinct features (or code units) in product family

The product family used in this evaluation consists of nine Linux kernel product variants. These product variants have in total 3146 features and 342729 code units (functions and data structures) (see Table 5.2). However, not all these features (or code units) are unique in the product family, because these product variants share common features and implementations.

Figure 5.4 presents the analysis results of distinct features in this family of nine product variants. The vertical axis represents the number of distinct features. The horizontal axis represents the number of corresponding features, represented by a distinct feature, across product variants. Distinct code units in the product family manifest the similar distributions (see Figure 5.5).

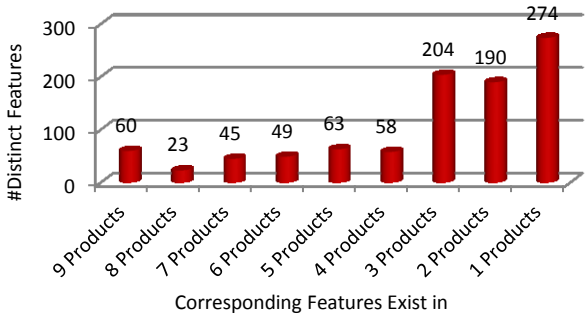

Figure 5.4. Distinct features of Linux kernel product variants

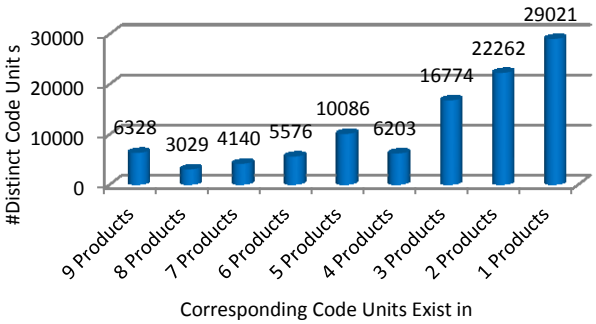

Figure 5.5. Distinct code units of Linux kernel product variants

Distinct-feature (or code-unit) analysis identifies 966 distinct features and 103419 distinct code units in the product family. That is, by identifying distinct features (or code units) in the product family, we can greatly reduce search space for feature location, from dealing with 3146 features of individual product variants to 966 distinct features of the product family (i.e. reduced by 69.3% at feature level), and from 342729 code units of in-

dividual product variants to 103419 distinct code units of the product family (i.e. reduced by 69.8% at implementation level).

Let us take a closer look at distinct features. 28.4% (274/966, the rightmost bar in Figure 5.4) of distinct features represent features that are supported by only one of nine product variants. These uniquely supported features are optional features for an operating system. For example, only the product variant *2.6.31.9.r1* supports the feature "LBDAF support for large (2TB+) block devices and files". The rest of 71.6% of distinct features represent features that are supported by at least two product variants. Among these distinct features, 60 distinct features (the leftmost bar in Figure 5.4) represent features that are supported by nine product variants. These commonly supported features usually deal with core architecture and algorithms of an operating system. For example, the core feature *CRYPTO_MD5*, described as "MD5 message digest algorithm (RFC1321)", is supported by each product.

5.5.3 Disjoint, minimal feature (or code-unit) partitions

Using FCA to group 966 distinct features and 103419 distinct code units among nine product variants generates 109 disjoint, minimal feature and code-unit partitions. Figure 5.6 and Figure 5.7 summarize the number of partitions (vertical axis) that have different number of distinct features and code units (horizontal axis).

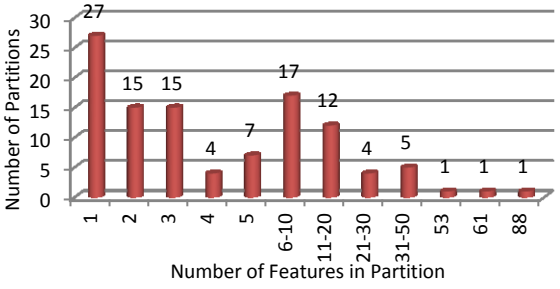

Figure 5.6. Partition size by features

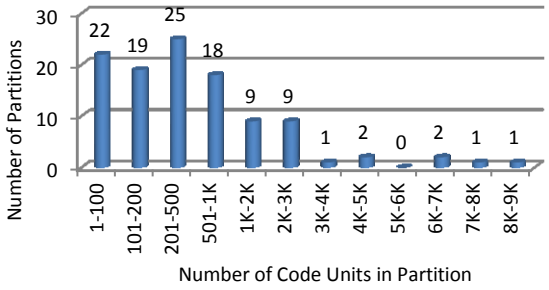

Figure 5.7. Partition size by code units

On average each partition consists of about only 1% (8.9±14, average±standard deviation) of all distinct features, and about only 1% (948.8±1540) of all distinct code units. 78% of these 109 partitions (the sum of the six leftmost bars in Figure 5.6) consist of 10 or fewer features; 77.0% of these 109 partitions (the sum of the four leftmost bars in Figure 5.7) consist of 1000 or fewer code units. This shows that by grouping features and code units into disjoint, minimal partitions we can further significantly reduce search space for feature location.

For example, the smallest 27 partitions consist of only one feature (the leftmost bar in Figure 5.6) and 2 – 640 code units. These 27 partitions represent features and the relevant implementing code units that are perfectly isolated by analyzing commonalities and differences of product variants. Even the two largest partitions (the two rightmost bars in Figure 5.6 and Figure 5.7) consist of 88 features and 7784 code units, and 61 features and 8249 code units, respectively, i.e. less than 9% of all distinct features and less than 8% of all distinct code units.

5.5.4 Performance of our FL-SPF approach

LSI groups related terms into topics based on their co-occurrence in the documents in a corpus. A major control parameter to LSI is the number of topics that should be used for

topic-model construction. We want enough topics to capture real term relations, but not too many, or we may start modeling irrelevant details in the data [41].

As a result, in this work we cannot use a fixed number of topics for LSI analysis, because we apply LSI to feature (code-unit) partitions of different size. Thus, we use a factor p_d ($0<p_d<1$) and determine the number of topics that should be used by $p_d \times Min(term_d, doc_d)$, where $term_d$ and doc_d are term and document dimensionality of term-document matrix generated by LSI. We examine the performance of our FL-SPF approach at p_d=0.1, 0.2, 0.3, 0.4, and 0.5.

In this evaluation, we focus on the top 30 code units returned by feature queries. This decision was mainly driven by the fact that the majority (about 70%) of Linux kernel features declares 30 or less functions and/or data structures. Furthermore, similar to other studies [149], we believe that developers would highly unlikely to examine low-ranking code units.

Figure 5.8 presents PRQ values for top 10, 20 and 30 returned code units at different p_d. At p_d=0.1, FL-SPF returns at least one relevant code unit in top 10 results for 48.3% of feature queries (i.e. 467 out of 966 distinct features). It returns relevant code units for 55.8% feature queries in top 20 results, i.e. 7.5% more relevant feature queries if we examine the 11th – 20th returned code units. It returns relevant code units for 5.6% more feature queries if we further examine the 21st – 30th returned code units. Clearly, our approach can return at least one relevant code unit in top 10 results for almost half of the features to be located. Examining more returned code units may find relevant code units for more features. However, the chance becomes less towards lower-ranking code units. Similar phenomenon can be observed at other p_d values.

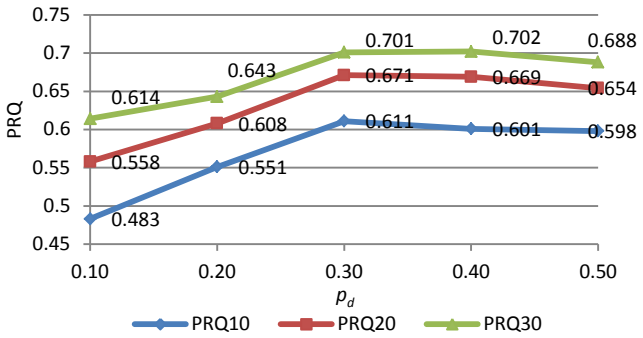

Figure 5.8. *PRQ* (N_q=10, 20, 30) at p_d=0.1,…,0.5

Figure 5.8 shows that PRQ values increase as p_d increases from 0.1 to 0.3. This can be attributed to the fact that more important term relations are captured as the number of topics to be modeled by LSI increases. As a result, our approach returns at least one relevant code unit for more feature queries. However, higher number of topics does not always result in better PRQ. PRQ values become stable at p_d=0.4 and start dropping at p_d=0.5. This is because LSI starts modeling peculiarities of data rather than important term relations in documents as the number of topics keeps increasing [41]. Consequently, it fails to return relevant code units for some features.

Table 5.3. MAP and *APCUI* (N_q=30) at p_d=0.1,…,0.5

p_d	0.10	0.20	0.30	0.40	0.50
MAP30	0.431	0.476	0.483	0.479	0.475
APCUI30	0.207	0.158	0.153	0.157	0.153

Table 5.3 presents MAP values and APCUI values for top 30 returned code units at different p_d. As p_d increases from 0.1 to 0.2, we achieve higher MAP value and lower APCUI value, which indicates that FL-SPF returns more relevant code units, and it ranks relevant code units higher in the results and thus less code units to be investigated before encountering the first relevant code unit. MAP and APCUI values remain stable at p_d>0.2, which shows that the impact of increasing p_d on MAP and APCUI becomes less significant at

p_d>0.2. However, our approach can still return relevant code units for more feature queries (as manifested by increasing PRQ value from p_d=0.2 to p_d=0.3 in Figure 5.8).

5.5.5 Comparison with direct application of LSI

We comparatively investigate the performance of our approach against that of the direct application of LSI to the dataset of nine Linux kernel product variants. We use 500, 1000, and 1500 as the numbers of topics for LSI analysis in the direct application of LSI. We evaluate the performance of direct application of LSI in terms of the average PRQ, MAP, APCUI values of LSI for each product variant.

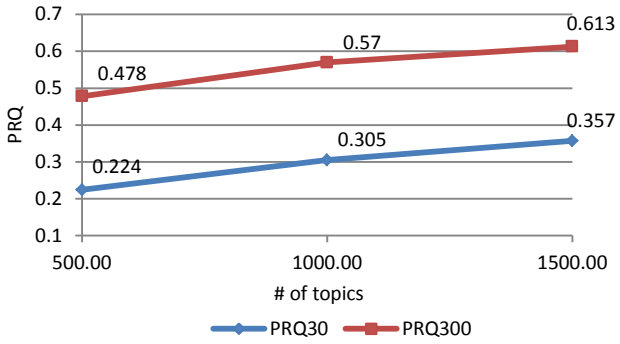

Figure 5.9. PRQ values of direction application of LSI

Table 5.4. MAP and APCUI of direct application of LSI

#Topics	500	1000	1500
MAP30	0.327	0.344	0.327
MAP300	0.115	0.133	0.14
APCUI30	0.292	0.284	0.283
APCUI300	0.236	0.202	0.182

Figure 5.9 presents the average PRQ values at 500, 1000, and 1500 topics. The best average PRQ value of direct application of LSI for top 30 returned code units is 35.7% at 1500 topics. This PRQ is much lower than the worst PRQ value of our approach for top 30 returned code units (61.4% for PRQ30 at p_d=0.1, see Figure 5.8). One way to improve PRQ is to increase the number of topics to be modeled by LSI. However, it is not always feasible to increase the number of topics for direct application of LSI, because it requires significant increase in memory and runtime for constructing topic models of the entire set

of features and code units of a product variant. In fact, our attempt to apply LSI at 2000 topics fails due to out of memory exception.

An alternative way to improve PRQ is to examine more returned code units. As shown in Figure 5.9, in order to achieve same-level PRQ value as our approach (61.4% for PRQ30 at p_d=0.1), we have to examine top 300 returned code units (61.3% at 1500 topics). However, as shown in Table 5.4, the improvement of PRQ from top 30 to top 300 returned code units comes with a significant degrade of the average MAP and APCUI values. The average MAP degrades from 32.7% for top 30 results to 14% for top 300 results; on average the number of code units that need to be investigated before encountering the first relevant code unit increases from 8.4 (28.3% of top 30 results) to 54 (18.1% of top 300 results).

In contrast, with our approach we do not need to pay this price. As shown in Figure 5.8 and Table 5.3, at p_d=0.1, our approach achieves PRQ=61.4%, MAP=43.1%, and APCUI= 20.7% for top 30 returned code units. Note that this is the worst performance of our approach. But the MAP for top 30 results of our approach is still 31.8% ((43.1%-32.7%)/32.7%) higher than that of top 30 results of direct application of LSI at 1500 topics. Furthermore, the number of code units that need to be investigated before encountering the first relevant code unit is about 8% (28.3%-20.7%) less with our approach.

We also measure the performance of our approach at p_d=0.1 for top 300 returned code units. The PRQ, MAP, and APCUI values are 80.6%, 24.3% and 11.1%, which means that, compared with top 300 results of direct application of LSI at 1500 topics, our approach returns at least one relevant code unit for 19% (80.6%-61.3%) more features, achieves 71.45% ((24.3%-14%)/14%) higher mean average precision, and requires investigating 21 less code units (|11.1%-18.2%|*300) before encountering first relevant code unit.

In addition to poorer performance, another disadvantage of direct application of LSI to each product variant is that feature location results in each product variant often vary greatly in terms of code units returned and their rankings. Take the Intel microcode feature as an example. It is supported in five out of nine product variants. Direct application of LSI returns relevant code units in only two product variants (fails to return relevant code units in the other three produce variants): two code units ranked 1st and 6th for product variant *2.6.37 r2*, and one code units ranked at 1st for *2.6.31 r2* in top 30 results. Given such feature location results, developers have to examine, merge, and propagate feature location results from each product variant in order to identify common implementations of a feature.

In contrast to dealing with features and code units of each product variant, our approach performs feature locations for distinct features and distinct code units in software product family. Once the link between a distinct feature and a distinct code unit is identified, the links between features and code units in each product variant can be easily inferred.

5.6 Threats to Validity

Our approach uses software differencing techniques to determine distinct features and code units in software product family. The quality of differencing techniques may affect our approach. Empirical studies [45,171,180] has shown the accuracy and robustness of differencing techniques to be good in practice. Thus, the negative impact of software differencing on our approach should be minor.

Our evaluation uses nine product variants of Linux kernel. Linux kernel manages features using Kconfig at feature level and using makefiles and C preprocessor at code level. However, it is important to note that our approach does not make any assumptions regarding how product variants are generated and managed. It requires as input only feature sets and program models of product variants.

Our approach assumes that commonalities and differences between product variants can be determined statically, such as product variants of Linux kernel used in our evaluation. 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 order to obtain feature sets and program models of different product variants at runtime.

Our approach partitions product variants through combination of their commonalities and differences. Further studies are required to investigate the impacts of the number of product variants and the differences among these product variants on the effectiveness of our approach.

5.7 Summary

In this chapter, we presented an approach to support effective feature location in software product families. Our approach effectively exploits software differencing and FCA techniques to analyze commonalities and differences of product variants in a software product family, which greatly reduce the search space to which IR technique applies. The evaluation of our approach with a product family of nine Linux kernel product variants showed that our approach always significantly outperforms a direct application of IR technique in the subject product family. This result suggests that we can significantly improve IR-based feature location in software product family by partitioning product variants. Our approach is at the core of our ongoing work on reengineering legacy software product family towards systematic reuse of overlapping feature across product variants.

6 Variability Management with Multiple Traditional Variability Techniques

In the previous Chapter 3, 4 and 5, we focus on the variability analysis in the domain engineering. In this chapter, we aim at investigating the management of variant features by using variability techniques to configure different variant features for different product variants in an industrial project [182].

6.1 Introduction

Even with identified variability between product variants available, the domain engineers still face the challenge of managing the variability properly towards the SPL infrastructure. How the variability is managed on earth in reality? To reveal the answer to the RQ4 mentioned in section 1.1, we conduct the empirical study in a software company based in Shanghai, China.

The goal of this chapter is to evaluate strengths and weaknesses of variability techniques used in the existing Wingsoft Financial Management System Product Line (WFMS-PL). This industrial SPL, some of whose features are exampled in section 3.1, 4.1 and 5.1, is developed by Fudan Wingsoft Ltd., a small software company in China.

We took the following steps in this study. We first analyzed WFMS variant features [73] and presented them as a feature diagram [83]. Then, we studied variability techniques in WFMS-PL, namely Java conditional compilation[3], commenting out feature code, design patterns [57], parameter configuration files, and a build tool Ant [188]. Finally, we analyzed how the granularity and scope of feature impact on WFMS components affected the effectiveness of variability techniques.

[3] Java does not formally have conditional compilation, but you can implement the similar function [197]: http://c2.com/cgi/wiki?ConditionalCompilationInJava

We distinguish two types of features according to the granularity of their impact, namely *fine-grained features* affecting many system components, at many variation points, and *course-grained features* whose code is usually contained in files that are included into a custom product that needs such features. *Mixed-granularity features* involve both fine- and coarse-grained impact. Most of the WFMS features were fine-grained features, managed with conditional compilation and/or manually commenting out the feature code.

Our study shows that different variability techniques have different, often complementary strengths and weaknesses, and their choice should be mainly driven by the granularity and scope of feature impact on product line components. Fundamental differences in capabilities of variability techniques justify the use of multiple variability techniques. For example, Ant is strong in configuring coarse-grained features, but weak in configuring fine-grained features. Parameter configuration files define environmental variables and variant feature options, but require yet other techniques to perform the actual customizations in product line components. Design patterns reduce the coupling in code, making it easier to add, remove or change a variant feature. However, we found only few opportunities to apply design patterns in WFMS. Over-loading fields in order to use the same field for different purposes usually helps only in configuring database schema. Conditional compilation is used as the main technique to control fine-grained variant features in Java source code, while commenting out feature code is heavily used in HTML and JSP files. In some situations, we suggest possible remedies to weaknesses of variability techniques used in WFMS-PL.

Particularly, multiple variability techniques must be used to manage each of the mixed-grained features. Our study reveals that while it is natural to match feature granularity with the proper variability techniques, over time the interplay between multiple variability techniques may be difficult to comprehend.

Variability techniques used in WFMS-PL are simple, freely available and commonly used in Software Product Lines (SPL) to complement component/architecture-based approaches. As yet we do not have enough material to compare them with more advanced SPL approaches, such as GEARS [190], Pure [199] or XVCL [71], with possibly better results. We are going to conduct experiments to facilitate such comparison.

In the past years, there were some case studies on variability techniques. These studies, however, usually focused on variation implementation with certain techniques like AspectJ [96], FOP [18,19,137] or XVCL [71]. The industrial case study presented in [160] aims at architecture-based variability realization in large companies. In this chapter, we analyze a real product line using a mixed set of light-weight variability techniques in a small company. We believe Wingsoft choice of variability techniques was representative of the variability managemet in the small companies. Some other small companies may find our experiences reported in this chapter useful, when migrating from existing product variants towards the product-line architecture.

6.2 An Overview of WFMS

WFMS was developed in 2003 and evolved to an SPL with more than 100 customers today, including major universities in China such as Fudan University, Shanghai Jiaotong University, Zhejiang University. During its evolution, Wingsoft set up product architecture and was adopting variability techniques such as Java conditional compilation, Ant, parameter configuration files, and design patterns to manage product variability.

The core assets of the WFMS-PL were designed and implemented by few *domain engineers*. Domain engineers sometimes also played the role of an *application engineer*s responsible for initial, program-level customization of core assets for a custom product. *Service engineers*, familiar with financial business but with little or no programming knowledge, did final customer-side customizations and deployment. Service engineers

used readable parameter configuration files to do the customer-side customizations. Usually, application engineers provided only in-office application-specific implementations, and responded to requests of service engineers. Domain engineers maintained a WFMS product for many customers delegating routine work to service engineers.

WMFS consists of four subsystems, namely Financial Management Subsystem (FMS), Salary Management Subsystem (SMS), Reward Management Subsystem (RMS), and Tuition Management Subsystem (TMS). We selected the TMS for our case study, as it involved types of variability and variability techniques that were representative of the whole WFMS. TMS is a web-based portal for students to pay online their tuition fee, with functions such as login, fee browsing, online payment, payment detail generation and bank settlement. The code of TMS is 25% of the whole WFMS system. It comprised 58 Java source files, 99 JSP web pages, and several configuration files.

A TMS feature diagram is shown in Figure 6.1. The minimum and maximum choices of OR-features are shown as numbers surrounded by square brackets. 80% of the 32 variant features can be selected for custom TMS. However, there are also some feature interactions. For example, the selection of *InitPayMode* depends on the number of selected variant features under *FeeItemSelection* and the selection of *Settlement* depends on whether selected banks require settlement. TMS features include fine-, coarse, and mixed-grained feature.

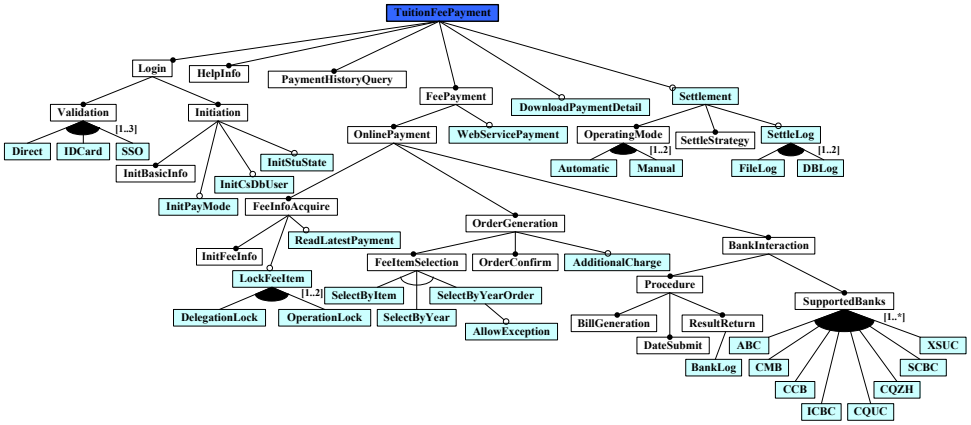

Figure 6.1. The feature diagram of TMS

As illustrated in Figure 6.2, the TMS is a web-based portal for students to pay their tuition fee online, with functions such as login, service customization, on-line payment and history query. In addition, the TMS also provides accounting services (e.g., report generation and bill settlements) that interface universities with banking systems. TMS adopts a traditional 3-tier architecture, namely user interface, business logic and database access tier.

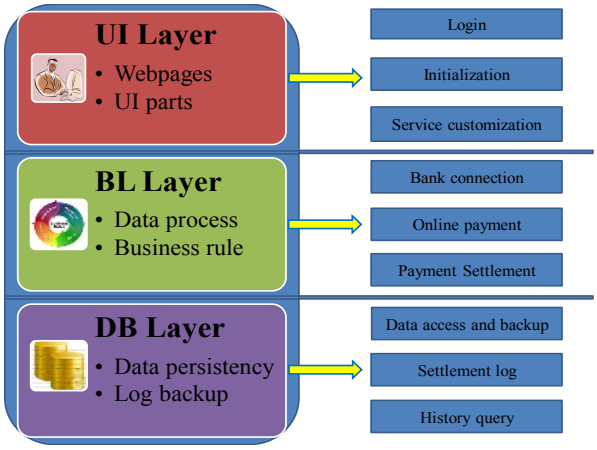

Figure 6.2. The architecture of TMS

Table 6.1. Variant features of TMS

TMS Feature Name	#VP	#Size	#AF	Description
Direct	1	7	1	The direct login mode
IDCard	3	11	1	The login mode with ID number
SSO	5	17	3	The single sign-on mode
InitPayMode	9	16	4	Initiation of the payment mode
InitCsDbUser	10	18	5	Initiation of user account of finance database
InitStuState	10	37	3	Initiation of the student's state
WebServicePayment	18	2144	17	Providing web service of payment
LockFeeItem	16	108	5	Locking fee items from being paid
DelegationLock	8	10	2	Locking fee items whose payments are delegated to bank
OperationLock	3	39	2	Locking fee items that are waiting for payment results
ReadLatestPayment	10	90	6	Showing the result of last payment
PayByItem	6	288	6	Selecting fee items by each item
PayByYear	6	176	6	Selecting fee items by the year of item
PayByYearOrder	6	603	6	Selecting fee items by the year order
AllowException	6	69	3	Allowing exceptions some pay mode
AdditionalCharge	7	15	3	Storing additional charge
ABC	18	1451	18	The Agriculture Bank of China
CCB	17	1043	17	The Construction Bank of China
CMB	11	784	11	The Commercial Bank of China
ICBC	24	1339	24	The Industry and Commercial Bank of China
CQUC	6	783	6	Office of Chongqing University in charge of fee payment
CQZH	8	1038	8	Chongqing Sub-Brance of the Bank of China
SCBC	7	546	7	Sichuan Sub-Branch of the Bank of China
XSUC	6	844	6	Office of Southwest Normal University in charge of fee payment
BankLog	13	111	11	Log interaction information with the banks
DownloadDetail	3	136	3	Downloading payment details
Settlement	13	487	13	Bank settlement
Automatic	4	216	3	Automatic operation mode
Manual	1	48	1	Manual operation mode
SettleLog	9	153	4	Log settlement info.
FileLog	7	103	6	Log information in files
DBLog	6	79	4	Log information in database

Table 6.2. Feature dependency and interactions

Involving features	Domain/Design level dependency	Implementation level interaction (in variation point)
ABC, ICBC, AdditionalCharge	If *ABC* or *ICBC* is selected, *AdditionalCharge* should be selected.	N/A
ABC, CMB, CQZH, Settlement	If *ABC* or *CMB* or *CQZH* is selected, *Settlement* should be selected.	N/A
InitPayMode, PayByItem, PayByYear, PayByYearOrder,	Number of the selected features among *PayByItem*, *PayByYear* and *PayByYearOrder* decides the selection of *InitPayMode*. If only one of them is selected, *InitPayMode* is not selected.	N/A
Direct, IDCard	N/A	1 vp in FeeUser.java
WebServicePayment, InitCsDbUser	N/A	1 vp in user-defined configuration file
WebServicePayment, PayByItem	N/A	1 vp in FeeOrder.java
ReadLastPayment, OperationLock	N/A	1 vp in DB schema
BankLog, DBLog	N/A	1 vp in DB schema

TMS components are derived from corresponding TMS core assets which have been instrumented with variability techniques to accommodate variant features required in custom products. As shown in Figure 6.1, in TMS feature model, there are 32 variant features and 9 mandatory features. Table 6.1 shows the impact of TMS features on core assets: #VP –

the number of variation points at which a feature affects core assets, #Size – LOC of feature implementation, #AF – the number of affected core assets. We see that most TMS features have fine-grained impact on core assets.

Feature dependencies constrain legal combinations of features that can be implemented in any custom product. For example, selection of one feature may entail selection of yet other features. In TMS, the feature *ABC* represents the system connected to the Agriculture Bank of China (ABC) and the feature *Settlement* means that a bank needs settlement for each payment. As feature *ABC* requires settlement for each payment therefore whenever we select the feature *ABC*, we must also select the feature *Settlement*. Inter-dependent features tend to interact with core assets at the same variation points, and may also affect each other code.

Table 6.2 shows that 18 out of 32 TMS variant feature involved in feature dependencies or interactions. The feature dependencies are imposed by the domain [51]: if the feature *ABC* is selected, the feature *AddiotnalCharge* should also be selected for that product. Feature interaction at the implementation level is due to certain design decision made during development or to the way that features are realized in the solution, e.g. two features' code tangle together in one method, one parameter/attribute in configuration- file/DB-Schema is shared by two distinct features. Both feature dependencies and interactions further complicate analysis, maintenance and reuse of both core assets and features. For example, if we use conditional compilation to manage the impact of feature interactions on core assets, then relevant variation points will include many options to choose from during customization, with each option catering for a specific type of feature interaction.

6.3 Variability Technique in TMS

Wingsoft adopted simple, freely available variability techniques for TMS-PL. Different variability techniques have different, often complementary, strengths and weaknesses, and

their choice is mainly driven by the granularity and scope of feature impact on core assets. At Wingsoft, experienced domain engineers were selecting the right variation technique for features at hand.

6.3.1 Review of variability technique in TMS

In Table 6.3, column "# Features" indicates the number of features whose customizations involved a given technique. Ant, conditional compilation and commenting out variant feature code were most commonly used. Note that we found overloading fields were only used for database tables, and managed variants of 13 attributes in 4 tables.

Table 6.3. Feature numbers for variability techniques used in TMP

# Techniques	# Features
Conditional compilation & comment	31
Ant	19
Overloading fields	13
configuration items	12
Design Pattern & reflection	3

Java conditional compilation and commenting out code: In Java, conditional compilation is realized with *final-boolean* variables. If a *final-boolean* variable's value is *false*, then the code in the statements under *if* is not compiled into the generated bytecode file. The effect is similar to *#define* and *#ifdef* C/C++ preprocessor directives [155]. Figure 6.3 illustrates the usage of *final-boolean* variables to manage variant features in TMS class *FeatureConfiguration*. The limitation of Java's conditional compilation is that it is limited to inner-method statements. It cannot handle the inclusion or exclusion of class methods or attributes. In WFMS, such cases were handled by manually commenting out the code that was not required in a given product variant. Commenting out was also used in non-Java files such as JSP files or SQL script. The main reason for such practice was that the engineers at Fudan Wingsoft could not find flexible tools to manage variability in these files at that time.

```
1   public class FeatureConfiguration {
2    // Configuration items
3    public static final boolean DelegationLock = true;
4    public static final boolean OperationLock = true;
5   }
```

```
1   public class FeeInfo {
2    . . .
3    public void initInfo(FeeUser user, boolean isPaidFeeInfo)
         throws Exception {
4        //get each year's fee items
5        for( int i=0; i < yearTemp.size(); i++ ) {
6            If ( FeatureConfiguration.DelegationLock
                 && FeatureConfiguration.OperationLock )
7                // Code when both features are selected
8            else if ( FeatureConfiguration.DelegationLock )
9                // Code when delegationLock is selected
10           else if ( FeatureConfiguration.OperationLock )
11               // Code when operationLock is selected
12       }
13   }
14  }
```

Figure 6.3. Managing variant features with Java's *final-boolean* mechanism

Design patterns and reflection: [111] has described the use of the abstract factory pattern in SPL. It also extended this concept using the *dynamic abstract factory pattern*, in which concrete factories were adapted to support new concrete products at run-time by adding *Register* and *UnRegister* operations to Abstract Factory for each abstract product. Although the most frequently used design patterns in TMS were *AbstractFactory* with *FactoryMethod* and *Strategy*, reflection mechanism, instead of operations returning name of product, is also used to dynamically instantiate proper concrete instances according to configuration options so that the specific class names can be abstracted from the source code. Then Ant can be used to control the inclusion and exclusion of a strategy subclass. Figure 6.4 shows the use of *Strategy* Pattern in TMS.

```
1   public class FeeOrder {
2    private Initializer initializer;
3    public init(FeeUser user, FeeInfo info,
         HttpServletRequest request) {
4        Class c;
5        try{
6            c = Class.forName( user.getPayMode() );
7            initializer = (Initializer) c.newInstance();
8            initializer.init ( . . .);
9        } catch(Exception e ) {
10           e.printStackTrace();
11       }
12   }
13  }
```

Figure 6.4. Reflection used in strategy pattern

Overloaded Fields: Variant features affect TMS database schema. Overloading table fields helps to contain some of those impacts. For example, a table may have several fields named *spec_1*, *spec_2* ... *spec_n*, and the same filed may be used to store bank card number in one product variant and ID card number in another one. There were also tables and fields that make sense for some product variants, but are useless for others. With overloading fields, all the products could share the same DB schema, but still support different data structures required for variant features. Overloading fields was adopted for WFMS-PL database.

```
1   <project name="webfee"  basedir="." default="main">
2   <target name="copy-src" depends="create-folders">
3     <!-- Copy java classes of Feature  PayByItem -->
4     <copy todir="${src.dir}">
5       <fileset dir="${core-src.dir}/${ PayByItem }"/>
6     </copy>
7   </target>
8   <target name="copy-webpage"
9     depends="create-folders">
10    <!-- Copy webpages of Feature PayByItem -->
11    <copy todir="${web-root.dir}">
12      <fileset dir="${core-webpage.dir}/${PayByItem }"/>
13    </copy>
14  </target>
15   <project>
```

Figure 6.5. Using Ant to include optional features

```
1   <webFee>
2    <paymode>PayByItem</paymode>
3    <bank-info>
4      <bankList>
5        <bank>ICBC</bank>
6        <bank>CCB</bank>
7        <bank>CMB</bank>
8      </bankList>
9      <ICBC>
10       <bankUrl>http: //mybank.icbc.com.cn/servlet/co..</bankUrl>
11       <keyPath>C: /apache-tomcat-5.5.25/webapps/...</keyPath>
12       <keyPass>12345678</keyPass>
13       <merchantid>440220500001</merchantid>
14     </ICBC>
15   </bank-info>
16   <DownloadDetail>true</DownloadDetail>
17   </webFee>
```

Figure 6.6. Using configurations files

Ant: An important class of configuration parameters was managed by Ant. [39] used Ant and configuration files to differentiate variability in process and variability in product. Lower layers always override the *build.xml* file of upper layers, by which only the *build.xml* of the leaf layer will take effect. By reading layer orders in configuration files,

the leaf layer can be determined. In WFMS, Ant was useful as a variability technique for coarse-grained variant features either. For instance, optional feature *PayByItem* of TMS was managed by Ant as shown in Figure 6.5. This feature was implemented by a Java class and a JSP file. The inclusion of this feature in a custom product was implemented by including the relevant files in the path of *javac* command in the Ant configuration file.

Parameter Configuration Files: In TMS, self-defined configuration files were also employed as a variability technique, as shown in Figure 6.6. Configuration files contained both data and control parameters. Data parameters, such as URLs of banking services and key path, are also widely used in single products. Control parameters were used to configure feature selections and usually worked together with other variability techniques. For example, the parameter *paymode* in Figure 6.6, indicating the right sub-class to be initialized, worked together with the reflection and the *strategy* pattern shown in Figure 6.4(see the underlined part). Another example of parameters working with other variability techniques is the parameter *DownloadDetail* in Figure 6.6. A simple tool was implemented to read this parameter and generate Ant script shown in Figure 6.5 if the value is *true*.

6.3.2 Summary of variability technique in TMS

Figure 6.7 shows which variant features were managed using which variability techniques. We do not show overloading fields which was used for variants in database table schema only and did not overlap with other techniques. More than 80% features (26 among 32) were managed by more than one variability technique: 13 features were managed by three techniques and three features by four techniques. Design patterns were always used together with other techniques. Another interesting observation is that almost all features involved the use of conditional compilation and/or commenting out feature code, as in WFMS, like in many other SPLs, we saw many fine-grained features.

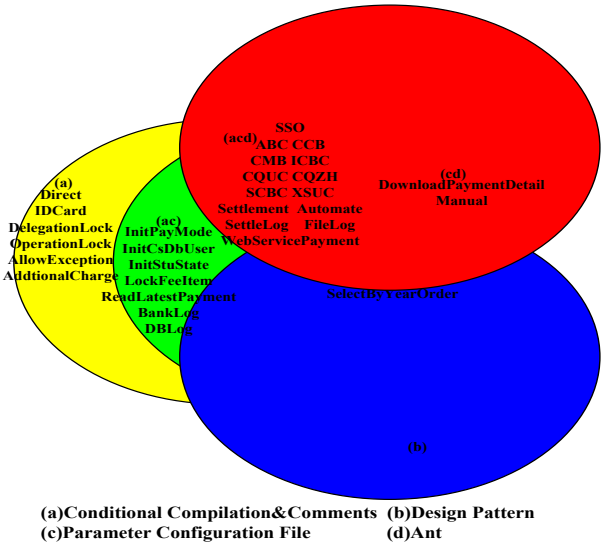

Figure 6.7. Variability techniques per feature

For mixed-granularity features, a combination of several variability techniques is usually used. For example, when we include a source files for a selected variant feature with Ant, we still need conditional compilation to configure corresponding caller in the base code. Table 6.4 summarizes the usage, scope, merit and drawbacks of various variability techniques used for different WFMS-PL variant features. It is interesting to have comparisons between the traditional approaches like the ones described in this chapter and more advanced ones like AspectJ [96], FOP [18,19] or XVCL [71]. In the next chapter, we discuss the XVCL-based solution of variability management in WFMS-PL.

6.4 Evaluation of the WFMS-PL and Possible Improvements

Projecting experiences from TMS study, we now evaluate WFMS-PL variability management strategies from the perspectives of granularity of feature impact, ease of applying, readability and managing the consistency between WFMS core assets and custom products.

On overall, we found that the current WFMS-PL strategies for variability management properly matched the granularity feature of impact. The newcomers to the team could easily understand when and how to apply them. However, over time when the impact of many

features was accumulating, the readability of the WFMS-PL has suffered, and it was becoming difficult to trace features to code and manage features consistently. The detailed reasons and suggested remedies are given in the following paragraphs.

6.4.1 Feature Granularity

Feature granularity is a critical factor that guides selection of variability techniques. It depends on the properties of the variant points, which we will elaborate in the next paragraph. In WFMS-PL, fine-grained features were managed by conditional compilation in Java code, and with commenting out code section in other WFMS artifacts. Ant was used to manage coarse-grained features at the level of package or class inclusion/exclusion level. Design pattern played the role of a class or method extension mechanism.

Table 6.4. Summary of variability technique in WFMS-PL

Variability technique	Usage	Scope	Merits	Drawbacks
Conditional compilation & Comments	Use *Final-Boolean* mechanism in Java or just simple natural language comments	*Final-Boolean* method is only used on inner-method statements. Comments can be adapted to all places	Easy to learn Straightforward No need to install and master new tools and skills	All maintain works and configurations are manual, no tool support. *If/elseif* statements are exponential with the number of possible variants
Design Pattern	To gain good modularization in OO source code,	Class or method level Best to gain class level flexibility in OOP part of a product line.	Providing Elegant code, High readability, Good Extendibility.	Scope of application is narrow and always need the aids of other techniques.
Overloading Fields	Make all customized products share the same attribute in database	Database table schema	Avoid trouble to change the name of attributes when selecting various products.	Hard to maintain, if the corresponding document is not available, very difficult to guess the meaning of the overloading fields.
Configuration File	Implement configuration of various parameters due to variant feature selection or environmental change	Give parameters of variant feature selections or environment	Good mechanism to do feature configuration	It needs to cooperate with other methods and introduce non-traceability issue. For features having fine-grain implementation fragment, configuration file is always not sufficient
Ant	Conditionally compile java source files and make deployments	System level customization and deal with more than source code), can only customize CGI features.	Powerful and popular build tool, flexible to deliver product variants	The finest granularity for ant is file level, so scope of application is narrow.

In Table 6.5, we show the number of WFMS variation points for each feature impact granularity level. Fine-grained features, those features with fine impact, required small changes in Java expressions, statements, method signatures, comments (in Java code, JSP or HTML), database table scripts, and parameter configuration files. Medium-grained fea-

tures required changes of Java methods or attributes, changes of database table scripts, and changes of configuration items in parameter configuration files. Coarse-grained features required inclusion or exclusion of product-specific source files.

Fine-grained features trigger most of the problems. Conditional compilation and commenting out feature code was used to manage fine-grained features. A big problem is how to trace variant features down to the many variation points relevant to them. This problem aggravates when multiple variability techniques are used to manage a given feature. Some feature enhancements involve changes at many variation points that must be properly coordinated. WFMS engineers often encountered the problems of inconsistent product release, e.g., a product variant was deployed with incorrect database schema.

Fine-grained and coarse-grained features were most common. We try to analyze the reasons as follows: Coarse-grained features are easy to configure. Whenever possible, domain engineers tried to contain the variant feature code in separate files which could then be included into custom products that required those features. Wizards could be implemented to allow application/service engineers to easily include such features into custom products. Fine-grained features are attributed to the variability of the business flow or logic of the application itself and the inner incapability of programming languages.

However, fine-grained features had to be accounted for the difficulty of consistent configuration. They often affected coarse-grained features and the new variation points are injected into the source code of course-grained features.

Table 6.5. The number of variation points per impact granularity level

Granularity	#Java	#JSP	#Conf. File	#DB Schema	#Total
Finest	14	0	0	0	14
Fine	67	43	3	3	116
Medium	18	0	7	5	30
Coarse	40	57	9	0	106
#Total	139	100	19	8	266

6.4.2 Ease of application

Customizations of core assets by configuring parameters and database schemas could be managed by service engineers who were in charge of deployment of a custom product on the customer site. Service engineers were familiar with general financial domain, user requirements, and basic deployment operations, but did not know much about programming and internals of the WFMS-PL. Wizard-supported parameter configuration files provided an easy-to-use configuration capability for service engineers.

Ease of application without involvement of any unconventional or proprietary techniques was the most important reason for Fudan Wingsoft to adopt simple and commonly available variability techniques for WFMS-PL. This reduced the learning curve and the staff training cost, important factors for any small or middle-sized company. Unless current techniques were found totally ineffective, Wingsoft would be chary of adopting new ones. WFMS-PL was constructed in lightweight, reactive way, which was in line with the company's benefits so far.

6.4.3 Readability

Readability refers to the reading ease of the code for the programmers. Design patterns and Ant did not hinder readability, but conditional compilation, commenting our feature code and overloading fields made code difficult to understand for applications engineers and even domain engineers. In our project, 30% of code in class *FeeOrder,* 20% of code in *FeeInfo* and 35% of code in *FeeUser* was managed by Java conditional compilation. Given that there are no other techniques to manage fine-grained features, this problem is very hard to solve. If we keep the code of variant feature code embedded in the base code, the code is bound to become hard to read.

One can consider Aspect-Oriented Programming (AOP) [96] or Feature-Oriented Programming (FOP) [137] tool AHEAD [16] to separate features from the base code, but

these approaches pose new problems as demonstrated in [89] and [90]. To improve the readability of the code, a promising approach is to resort to visualization tool's support such as CIDE [90] or XVCL-based Pre-processing [75, 76].

6.4.4 Traceability and extensibility

Traceability between features and their respective variation points has to do with both feature reuse and evolution. Here are some examples of problems:

- Each feature may be addressed at many variation points scattered through many SPL core components. To reuse or modify the feature we must find and analyze code at all these points.
- One SPL core component is usually affected by many features that may be managed by different possibly overlapping variability techniques. To reuse or modify the feature we must understand interactions among these techniques.

Variability techniques described in this chapter provide a workable but not perfect solution for traceability problems. Table 6.6 shows features that involved several variability techniques, with their respective variation points spread across different WFMS-PL core components. How to manage these variation points consistently was the issue of traceability. The difficulty in traceability also brought in the problem of product extensibility at those variation points.

Table 6.6. The number of variation points in example features

Variant Feature	Preprocessing	Conf. Files	Ant	Total
WebService-Payment	6	2	2	10
ABC	2	1	3	6
CCB	1	1	2	4
CMB	2	1	2	5
ICBC	1	2	3	6

Even there is another kind of traceability problem, namely two-way propagation of changes, one of the problems that hinder reuse between SPL core components and custom-

ized product variants [133]. We believe that the general traceability problem is difficult to address in the frame of variability techniques described in this chapter. A meta-level representation of the SPL core components paves the way for more effective solutions to these problems [78]. They can capture and manage synchronously the overall impact of features on SPL core components [72].

6.5 Summary

In this chapter, we evaluated strengths and weaknesses of variability techniques used in the existing Wingsoft Financial Management System Product Line (WFMS-PL), developed by Fudan Wingsoft Ltd. Feature characteristics must be matched by the capabilities of a variability technique(s) used to manage a given feature. Fundamental differences in capabilities of variability techniques justify the usage of multiple variability techniques.

Our study confirmed that different variability techniques have different, often complementary, strengths and weaknesses. Their choice should be mainly driven by the granularity and scope of feature impact on product line components. In some situations, we suggest possible remedies to weaknesses of variability techniques used in WFMS-PL. Our study reveals that while it is natural to match feature granularity with the proper variability technique, over time the inter-play between multiple variability techniques may be difficult to comprehend.

Variability techniques used in WFMS-PL are simple, practical, commonly used in SPLs to complement component/architecture-based approaches. We hope our report will help companies to make more informed decisions when moving towards the product line approach.

7 Variability Management with Uniform Variability Technique---- XVCL

This chapter is based on [176] to answer the RQ5 introduced in section 1.1. The investigation summarized in Chapter 6 at Fudan Wingsoft Ltd revealed potential scalability problems of multiple variability techniques. As a remedy, we replaced multiple variability techniques originally used in WFMS-PL, with a single, uniform variability technique of XML-based Variant Configuration Language (XVCL). This chapter provides a proof-of-concept that commonly used variability techniques can indeed be superseded by a subset of XVCL, in a simple and natural way.

7.1 Introduction

In previous chapter, we analyzed WFMS-PL, developed by Fudan Wingsoft Ltd in Shanghai. WFMSes provide web-based financial services for employees and students at universities in China. Following a common practice, Wingsoft set up product architecture, identified core assets for reuse, and then applied a range of common design-time variability techniques, such as conditional compilation, commenting out feature code or configuration parameters, to manage product-specific features in core assets. Variability techniques mark variation points in core assets to help developers perform customizations, manually or sometimes in an automated way.

The preliminary results from WFMS-PL case study showed that coarse-grained features are easier to manage than fine-grained features [182]. Feature granularity depends to some extent on the design of core assets. Good architectural design can change feature granularity in our favor, increasing the number of coarse-grained features, and reducing the number of variation points in core assets for the features that remain fine-grained. However, the orthogonal separation of concerns (or features) to get the prefect modularity without fine-

grained features is not always feasible [161]. Thus, proper variability techniques are more desired and important to manage features that still remain fine-grained or mixed-grained.

As shown in previous chapter, variability technique must match the feature granularity. Therefore, it is common to use multiple variability techniques, for example, conditional compilation to handle fine-grained features and a build tool such as Ant to handle coarse-grained features. Such variability techniques are easy to apply, and most of developers are familiar with them. However, as our study revealed [182], applying multiple variability techniques does not scale well, especially in cases of fine-grained features. While reuse and modification of fine-grained features is inherently difficult, applying multiple, often poorly compatible variability techniques aggravate the problems. In particular, it becomes increasingly difficult to find and understand already scattered feature code, and to coordinate changes required at multiple variation points.

As a remedy to the above problems, in the follow-up study we replaced variability techniques originally used in the Fudan Wingsoft product line, with a single, uniform variability technique of XVCL (XML-based Variant Configuration Language) [71]. XVCL applies generative mechanisms to organize software into highly some parameterizable meta-components. These meta-components form SPL core assets that are adaptively reused in product derivation, the process that is automated by the XVCL Processor [72]. This chapter serves as a proof-of-concept confirming that commonly used variability techniques can indeed be superseded by a subset of XVCL, in a simple and natural way. We also present an initial evaluation of benefits and trade-offs involved in adopting a uniform variability technique.

A practical lesson learned from our study is that in small- to medium-size product lines, applying multiple variability techniques may be a viable solution, as it requires less training, and variability can still be effectively managed in that way. As the product line grows

in size and the impact of features on core assets becomes more complex, a company may experience problems. Then moving towards a uniform variability technique approach may be beneficial. However, this will require a more systematic approach to reuse, and training of SPL personnel.

7.2 Problem of Adopting Multiple Variability Techniques

Fine-grained features are the main source of problems for scalability of the multiple variability techniques to managing SPL variability. Scattered impact of fine-grained features brings forth the difficulties to keep multiple variability techniques in synchronization one with another.

Feature *PayByItem* in Figure 6.4, Figure 6.5 and Figure 6.6 illustrates the problem. Inter-related configuration parameters control both Ant and Java conditional compilation. If the payment mode is switched from *PayByItem* to *PayByYear,* then the Ant script must be changed accordingly, and variation points controlled by Java conditional compilation, the commented out code in DB scripts and JSP scripts have to be modified accordingly.

As can be seen in Figure 6.7, 26 among 32 variant features were managed by more than one variation technique, 13 features - by three mechanisms, and 3 features - by four mechanisms. There are many examples of such interactions between variability techniques like that of Feature *PayByItem* in the original TMS core assets. Its maintenance and evolution entails the accurate understanding of multiple variability techniques, and familiarity with variant features and core assets.

As the size of the system grows and the feature dependencies increase, the above inconveniences aggravate. These observations encouraged us to experiment with a strategy that employs a single variability technique with capabilities to manage all the variability situations found in TMS core assets in a uniform and traceable way.

7.3 Single Variability Technique Approach to TMS Core Assets

XVCL [71,72,205], based on Frame Technology [15], is a generative language-independent variability technique for SPLs.

7.3.1 Variability technique of XVCL

XVCL encapsulates core assets in meta-components called *x-frames*. Each variation point in core assets is marked with a suitable XVCL command, such as **<adapt>**, **<insert-before> <insert>**, **<insert-after>** and **<break>**, to enable customizations. SPL variant features are formally mapped into all the relevant variation points in core assets by means of XVCL parameters and commands. The SPeCification x-frame, called **SPC**, sets values of XVCL parameters according to feature selection. XVCL Processor interprets x-frames starting from the **SPC** (Figure 7.1), traverses x-frames, propagates customization information (parameters) to them, adapting visited x-frames accordingly, and emitting code for a custom product. XVCL mechanisms allow us to manage features with fine-, coarse- and mixed-grained impact on core assets. Due to its language-independence, any type of SPL core assets including Java code, JSP files, DB scripts, textual documents (e.g., in a textual file), test cases or even UML models in XMI can be consistently customized for any legal selection of features required in a custom product.

7.3.2 TMS core assets instrumented with XVCL

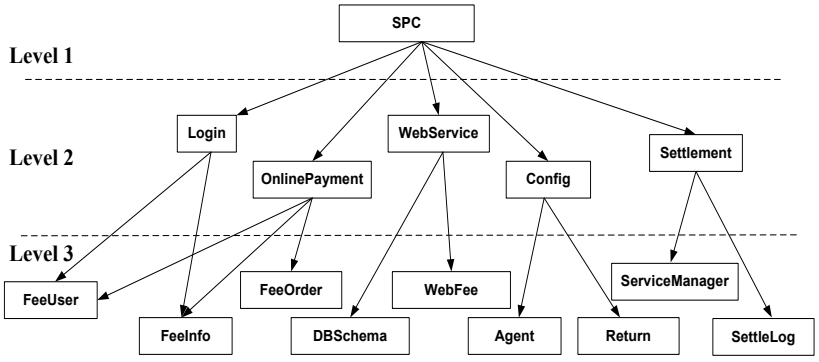

Figure 7.1. Overview of WFMS core assets in XVCL

Figure 7.1 provides a snapshot of the WFMS core assets in XVCL representation, and Figure 7.2 expands some x-frames to highlight the working mechanism of XVCL. The **SPC** specifies which features we need in a custom WFMS product by setting values for XVCL parameters that correspond to selected features. Values of those parameters propagate to x-frames below, navigating configuration and detailed customizations of core assets and features accordingly. **Level 2** x-frames define architecture-level customizations, in terms of configuration of core assets for a custom WFMS product. Some of the coarse-grained feature impacts are also addressed at Level 2. **Level 3** x-frames contain the actual code of core assets and features, instrumented with XVCL commands to enable customization of fine-grained features.

Features that we want to select for a custom product are assigned non-empty string values, while features to be de-selected are assigned empty string values. Therefore, **SPC** shown in Figure 7.2 selects features *IDCard* and *SSO* (related to *Login*), and feature *PayByItem* (related to *Paymode*) for a custom product. It deselects feature *Direct*, *PayByYear* and *PayByYearOrder*.

<**select**> commands mark variation points in x-frames below **SPC**. The value of an XVCL parameter that controls <**select**> identifies an <**option**> to be processed. <**select** PayByItem> in x-frame **OnlinePayment** at Level 2 illustrates a simple variation point affected by one feature only, namely *PayByItem*. If feature *PayByItem* is selected, then the Processor emits feature code to the custom product; otherwise, <**select**> has no effect.

<**select** Login> in x-frame **FeeUser** at Level 3 marks a variation point affected by three features, namely *IDCard*, *SSO* and *Direct*. Notation **@v**, where v is an XVCL parameter, means reference to v's value, as assigned in respective <**set**> command. The value of Login, <**set**> to be a concatenation of the three XVCL parameters corresponding to these features, controls <**select**>, directing processing to the <**option**> corresponding to the particu-

lar combination of selected features. Note that <**option** "IDCard+SSO"> is processed whenever the two interacting features *IDCard* and *SSO* are selected. Symbol '+' is a separator.

XVCL parameters formally link together customizations of all the core assets affected by selected features, at all the relevant variation points. XVCL parameters set in **SPC** create a bridge between features and WFMS core assets in XVCL.

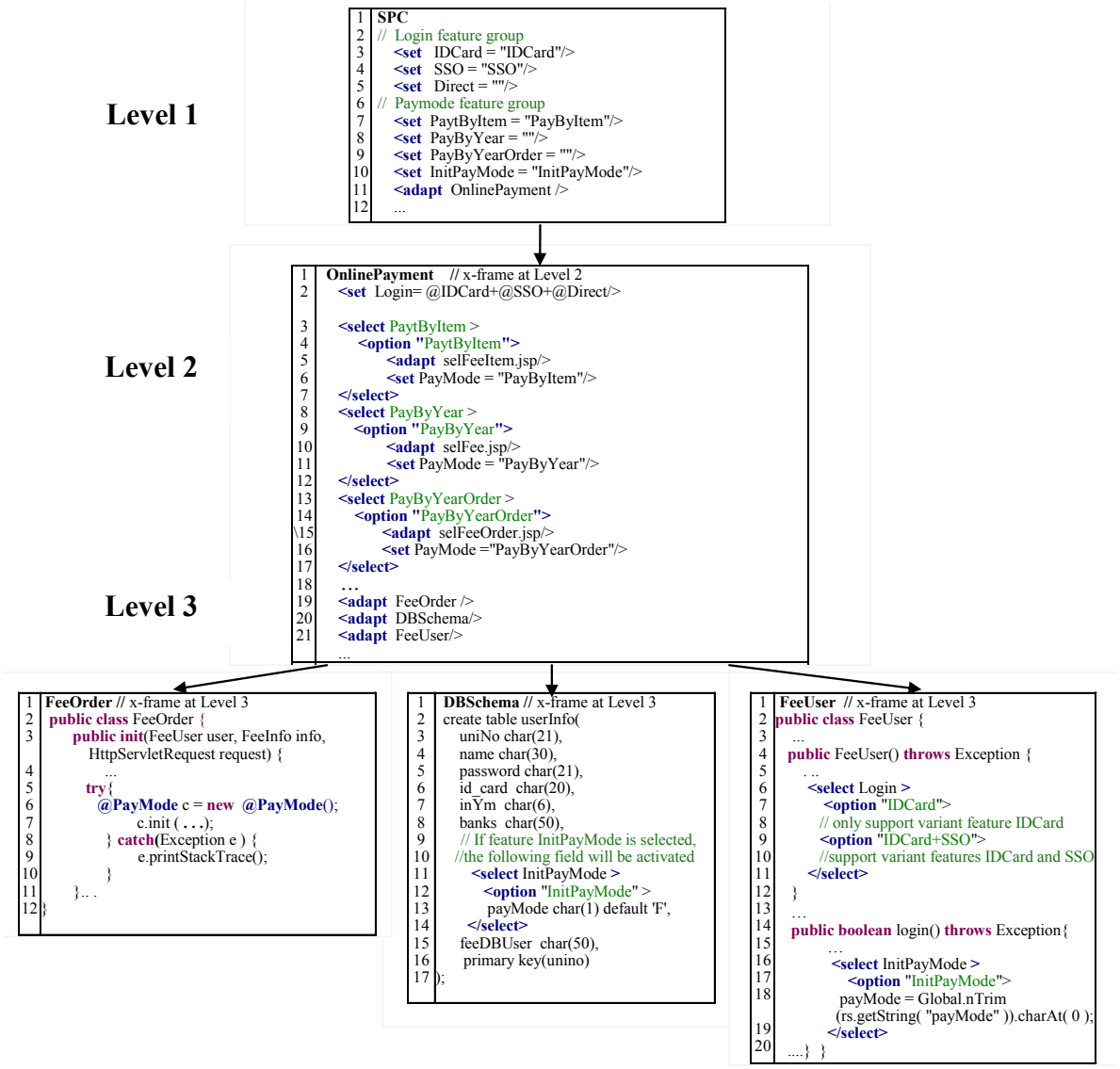

Figure 7.2. Detailed view of WFMS core assets in XVCL

7.3.3 One variability technique instead of many

Our example of Figure 7.2 also illustrates how a single variability technique can successfully provide capabilities of many variability techniques. In x-frame **FeeOrder,** **@PayMode** c = **new @PayMode** replaces configuration files and applications of *Strategy* pattern in the original WFMS core assets in Figure 6.4, Figure 6.5 and Figure 6.6. Here, we need a parameterizable name of the class. Java generics support parametric types, but not class names. In the original WFMS core assets, *Strategy* pattern and configuration parameter stored in a configuration file were used to mitigate the problem (Figure 6.4). *Strategy* pattern reads the name of a required class from the configuration file.

In the original WFMS core assets, architecture-level configurations of core assets and coarse-grained features were done by Ant. For example, if we select feature *PayByItem,* Ant's command <**fileset dir>** in Figure 6.5 includes file selFeeItem.jsp into the custom product. The same is achieved by **<adapt>** placed under **<select>** in x-frame **OnlinePayment**. Of course, Ant has more capabilities than XVCL's <**adapt**>, but in this context only Ant's asset configuration capabilities are used.

In x-frames **DBSchema** and **FeeUser**, we see how XVCL's parameters and <**select**> replace conditional compilation and commenting out feature code. For example, feature *InitPayMode* affects **DBSchema** and **FeeUser** and is managed by conditional compilation and commenting out technique. Manual modification of the conditional compilation or comments has to been done to include/exclude features. In XVCL on the other hand, variation points are inter-lined and customizations are automated.

7.3.4 Feature queries

Variability management with single variability technique also has the merits that tools can be implemented to help developers analyze, reuse and maintain features and core assets. To reuse or maintain features, developers must be able to locate and analyze all the

variation points at which features affect core assets and each other. Variation points for fine-grained features spread through many core assets. Using a single, uniform variability technique such as XVCL allowed us to implement query tools that can help in feature code location and analysis.

Developers specify features of interest in FQL (Feature Query Language) [75,76]. The tool evaluates queries and displays the results. In FQL, we can ask queries such as "which base components are affected by feature *f* and at which variation points?", "which features interact with feature *f*?", "in which base components and at which variation points feature f_1 interacts with feature f_2?". FQL is an SQL-like notation. We write queries in terms of XVCL elements.

Figure 7.3 shows a query to locate all the variation points at which feature *InitPayMode* affects WFMS-PL core assets.

```
declare x-frame x; option o;
select x, o
where o.f-names = "InitPayMode"
      and Contains (x,o)
```

Figure 7.3. Finding code of feature *InitPayMode*

Figure 7.4 shows a query that finds all the variation points at which features *IDCard* and *SSO* interact one with another.

```
declare x-frame x; option o
select x, o
where o.f-names="*IDCard*SSO*"
      and Contains (x,o)
```

Figure 7.4. Finding feature interactions

We refer the reader to [75,76] for details of query-based feature analysis.

7.4 Evaluation

What is the impact of replacing multiple variability techniques with a single one on SPL productivity? To answer this question we conducted lab studies and collected inputs from Fudan Wingsoft Ltd. regarding the original WFMS core assets developed by Wingsoft using multiple variability techniques, and core assets in XVCL. Below, we comment on productivity during domain engineering (i.e., building and evolving core assets), and product derivation.

7.4.1 Domain engineering effort

The original WFMS core assets were built by gradual re-engineering of existing WFMSes. Core components and their interfaces were stabilized first, and then variability techniques were used to prepare them for ease of customization, as described in chapter 6. While it is difficult to precisely determine the effort to build core assets, we obtained some relevant information from Wingsoft engineers who were involved in re-engineering. Selecting suitable variability techniques for various features was not difficult for experienced engineers. Also, each step of applying variability techniques was quite simple. New staff joining the Wingsoft team had no difficulty to understand the variability techniques used in WFMS core assets and their role. Some problems could be observed during evolution of the WFMS core assets. When multiple variability techniques were used together to accomplish a variability task, it might not be clear how to find all the relevant variation points, and understand the exact interplay between variability techniques. Still, given the size of WFMS core assets and relatively small number of features, the solutions adopted by Wingsoft team were considered to be adequate for the purpose.

To get insights into the effort of replacing multiple variability techniques with XVCL, the author and one engineer re-engineered the original WFMS core assets into XVCL representation. The engineer from Fudan University was a WFMS expert, also participating in

maintenance of the original WFMS core assets. It took two weeks for them to replace multiple variability techniques with XVCL in core assets for TMS subsystem. Applying XVCL was greatly simplified, as core assets were already in place, and they preserved most of the variation points. The main task was to work out overall XVCL controls and then to replace multiple variability techniques with XVCL commands at respective variation points.

Evolution of core assets involves adding new features and modifying features. The effort to evolve core assets depends on the number of variation points involved in change, and the complexity of finding, analyzing, changing variation points and tracing the impact of change. While the number of variation points in both solutions is almost the same, we assume that evolution of XVCL solution is easier than evolution of the original solution. This is due to uniform treatment of features, formal links between all the variation points relevant to a given feature, and feature query system.

7.4.2 Product derivation and maintenance effort

Deriving new products includes reuse of existing features, modifying features, and implementing extra features required by customers. As before, the effort of each such task depends on the number of variation points involved in change, and the complexity of finding, analyzing, changing variation points and tracing the impact of change.

Table 7.1 summarizes statistics relevant to product derivation effort. “Managed variation points” means variation points that have to be revised manually when reusing or modifying features. “Managed variation points” is a subset of all the variation points at which one feature affects core assets. For example, among core assets affected by feature *InitPayMode* are Java files and DB schema files. To reuse this feature in the original WFMS-PL, all affected files need to be manually changed. In the XVCL solution, once we <**set**> value of XVCL parameter InitPayMode in **SPC** (Figure 7.2), all the customizations for feature

InitPayMode spark from there, can be found by feature queries, and automatically performed by the XVCL Processor. Feature *InitPayMode* requires only 1 managed variation point.

Table 7.1. Managed variation points

	#variation points	#managed variation points	# files containing managed variation points
Original WFMS core assets	275	126	31
XVCL WFMS core assets	275	40	6

As another example, core assets affected by feature *Settlement* include seven Java files, four JSP scripts, one configuration file, and one file containing DB schema. To reuse feature *Settlement* in the original WFMS-PL, we must change code at eight variation points handled by conditional compilation, comments and Ant. We have 13 variation points, and 8 managed variation points. The location of managed variation points as well as relationship among them is not formally captured, therefore must be communicated via external documentation or re-discovered when needed. In XVCL solution, for the same feature there are also 13 variation points, but only 3 managed variation points (XVCL parameters for *Settlement* and for two dependent features). All the variation points are inter-linked via relevant XVCL parameters <**set**> in **SPC**, and reuse of the feature is automated by the XVCL Processor.

7.4.3 Other inputs from Wingsoft

Comments on code readability. Both XVCL representations and the original *final-boolean* conditional compilation and commenting out applied variability techniques to embed fine-grained feature code in the code of core assets at relevant variation points. About 30% of code in class *FeeOrder*, 20% of code in *FeeInfo* and 35% of code in *FeeUser* was managed by *final-boolean* conditional compilation and commenting out (the similar percentage in XVCL representations). Wingsoft engineers were concerned about readability

and maintainability of the code, but they were also pleased with XVCL's ability to mark traces of customizations relevant to a given feature and ease of finding all the variation points relevant to a given feature.

Comments on copyright protection. In the original WFMS core assets, feature code is embedded in core assets, but only some features are needed in a custom product. Wingsoft engineers often included unnecessary feature code into a custom product because of time involved in feature removal, and also because such code might be useful in future enhancements of a custom product. When extra functionality is contained in files that are released in readable form (e.g., JSP or XML configuration files), this practice can sometimes create copyright problems, as other customers may use extra functionality that was not meant for them and they did not pay for. Such cases happened in Fudan Wingsoft experience.

In XVCL, unwanted features are never included into a custom product, as the job of feature manipulation is consistently and automatically done by the XVCL Processor. Other than protecting copyrights, such precise and flexible control over feature inclusion/exclusion to/from custom products also matters in situations when we need to build highly optimized products, for example embedded software.

7.4.4 Evaluation summary

Overall, it was felt that for small-to-medium systems such as WFMS (around 50KLOC), adopting multiple variability techniques is still practical. Variability techniques used in the original WFMS are simple and known to most of engineers. They came into engineers' mind naturally, could be applied on the fly during core asset design, with minimum disruption of conventional programming. Multiple techniques provide an elementary infrastructure for SPL support. Handled by the experienced engineers, the original WFMS core assets serve well for the derivation of almost 100 product variants.

As the size of core assets and the number of variant features grows, and feature interactions get more complicated, problems may show up. Feature reuse and maintenance may become more complex because of the many variation points at which feature code needs be understood. Manual customizations become time-consuming and error-prone, even for skilled domain engineers. Then, it may be worth to consider migrating to a uniform variability technique such as XVCL.

In XVCL, for a feature reused as-is we need small number of managed variation points, at which we <**set**> XVCL parameters for that feature and its dependent features (in **SPC**). All the variation points for a given feature are formally linked to XVCL parameter representing that feature. The ability, which locates and analyzes traces of customizations for each feature, helps developers reuse and modify features with less errors and unwanted side-effects as compared to working with the original WFMS core assets. Reuse is automated by the XVCL Processor.

However, the adoption of XVCL is not without pitfalls, some of which XVCL shares with other variability techniques. Much of the code of features still remains tangled with core assets, affecting readability. This is a big problem, but so far alternative approaches based on specification-based variation points such as AOP [96] or FOP [18] failed to provide an effective solution to fine-grained feature management in SPLs [89, 90] (we comment further in Section 7.5). XVCL's feature queries can help developers to identify variation points relevant to various features. However, the actual feature modifications are not easy if the number of variation points is large. Training must be provided for the team to learn a new technique. The correctness of transformations from XVCL representation to code can be checked only during compilation.

7.5 Related Work

Managing variability is the essence of software Product Line (SPL) practice [32]. Variability techniques are one of the enablers of reuse-based derivation of products from reusable core assets. Productivity gains due to reuse to much extent depend on the effectiveness of product derivation. The importance of having adequate variability technique fitting specific needs of given SPL is stressed by SPL proponents and practitioners [15,33,86,160]. Deelstra et al. [40] identify the weakness of variability techniques as one of the obstacles that impedes implementing reuse strategies via the SPL approach in some industries.

Karhinen et al. [86] analyzed problems of managing variability solely at the implementation level, for example, using conditional compilation or configuration management tools. Their experiences from Nokia projects indicate that managing features with **#ifdefs** while technically feasible, is inherently complex, error-prone and does not scale. They proposed to use design means to manage variability. Similar problems with conditional compilation were also reported in FAME-DBMS [143].

The architectural/component approach to SPL applies design means to manage variability, in the attempt to modularize features as far as this is possible. Still, in most of application domains many features remain fine-grained, with their impact scattered through core assets [86,90,182]. Such fine-grained feature must be managed with additional variability techniques such as described in Section 6.3 and Section 7.2.

Industrial tools such as GEARS [190] and pure::variants [199] could certainly manage the WFMS SPL. However, we do not have hands-on experience with those tools or specific studies to provide detailed comparison. GEARS can handle configurable software artifacts – such as source code, test cases and requirement documents. Guided by the *product feature profiles*, which model optional and varying feature for each product, its configurator automatically assembles and configures the software assets. Pure::variants captures the

problems (*feature model*) and the solutions (*family model,* which records the customized feature models for product variants) separately and independently, to reuse the solutions and of the feature models in new projects. Apart from the possibilities of the programming languages and tools in the original product, to generate the product variants it also provides additional possibilities like AspectC++ or PatternTransformer.

The research community proposed Feature-Oriented Programming (FOP) [18] as an approach to feature management for SPL reuse. FOP is based on feature modularization, and a mechanism for feature composition into a base program. One of the motivations of FOP is to support SPLs. Mixin technique [154] has been widely used for FOP, with AHEAD [16] being its most advanced realization. AHEAD provides powerful solution for feature management in many situations, but is not geared for fine-grained features.

A number of authors also proposed Aspect-Oriented Programming (AOP) [96] as a variability technique. Using AOP, features are modularized as aspects (advices and introductions) and then weaved (feature composition) into a base program. A recent study has revealed difficulties in using AspectJ as a FOP realization technique [89]. Kästner and his colleagues concluded that the difficulties are attributed to the essential complexity of Feature-oriented refactoring of legacy applications and the fragility of Aspect's point-cut. Furthermore, readability and maintainability of the resulting code is undermined by the basic capabilities of AspectJ, even not with conditional extensions or homogeneous extensions.

In view of the above findings, Kästner et al. [90] relaxed the requirement for feature modularization, and revisited the idea of keeping feature-related code together with the code of core assets. They proposed a tool called CIDE (Colored IDE) to visualize feature code in core assets. CIDE helps programmers find and manipulate feature code. As CIDE works on an abstract syntax tree, it cannot handle some fine-grained feature impacts. The granularity of CIDE is not as fine as that of C/C++ preprocessors. For example, according

to one return type per method in the AST syntax, developers cannot specify two alternative result types of a method. In contrast to CIDE, XVCL uses programming language-independent representations to achieve similar goals.

In CIDE, annotations may be validated to preserve language rules, while XVCL <**select**> commands may be placed in arbitrary program points, leading to syntactic errors in the generated code. Aligning XVCL representation with constructs of the underlying programming language is possible and can alleviate some problems, but aligning is not enforced by the method. In addition, in some cases XVCL deliberately breaks such alignment for better flexibility in feature management. Validating meta-level transformations is the strength of CIDE, while simplicity, language-independence and the ability to handle any variability situations are strengths of XVCL.

Apel et al. [5] further proposed a framework and tool chain, FEATUREHOUSE [194], which is a descendant of AHEAD program generator. Similar to XVCL, FEATUREHOUSE can serve as a uniform variability technique and support the composition of several different types of software artifacts. Yet different from XVCL's mixing of meta-level commands and artifacts, it uses the superimposition technique FSTCOMPOSER, which is based on a general model of the structure of software artifacts, called *the feature structure tree* (FST). Based on the superimposition or merging of FSTs, the corresponding artifacts are changed accordingly via the support of FEATUREHOUSE.

In FEATUREHOUSE, in virtue of the FST, the direct annotation in artifacts is avoided and the readability is not undermined. The trade-off is that it has to integrate the various adapters and computation rules for the different languages. Since no annotation inside the artifacts for feature code at arbitrary granularity, we find that FEATUREHOUSE has to adopt hook a method to deal with the fine-grained feature impact. Compared with XVCL,

FSTCOMPOSER in FEATUREHOUSE is flexible at supporting additional features. It cannot really change an existing fragment. But XVCL is flexible at in what can be variable.

7.6 Summary

This study was conducted jointly by Fudan Wingsoft Ltd., a software company based in Shanghai, researchers at Fudan University and National University of Singapore (NUS).

In this chapter, as a remedy to the scalability problem of multiple variability techniques, we presented an approach based on a single, uniform variability technique of XVCL, capable of managing both fine-grained features, as well as features whose impact requires customizations at the product architecture/component level. We evaluated the XVCL-based product line representation in lab experiments, and in Fudan Wingsoft Ltd, a company that initially used multiple variability techniques and then applied XVCL.

Overall, for small-to-medium systems such as WFMS (around 50KLOC), multiple variability techniques are still practical. Handled by experienced engineers, the original WFMS core assets serves well for the derivation of almost 100 product variants. Common variability techniques can be applied with minimum disruption of conventional programming. Multiple mechanisms provide an elementary infrastructure for SPL support.

However, as the impact of features on core assets accumulates and gets more complex, understanding and synchronizing multiple, poorly compatible variability techniques may become difficult. We may have much manual, repetitive and error-prone work to do during reuse and evolution of core assets and features.

This weakness of multiple variability technique approach is the strength of a uniform variability technique approach such as XVCL. XVCL captures customization knowledge in human-readable and machine-executable (by XVCL Processor) form. Therefore, feature reuse is simplified and automated. Knowledge of prior customizations helps in designing customization required by new features during SPL evolution. However, the design of re-

usable core assets with a uniform variability technique requires more skill and effort. Both common variability techniques and XVCL keep fine-grained feature code embedded in core assets which hampers readability. XVCL tried to alleviate this problem with feature queries that navigate developers to all the variation points relevant to features. This is only a partial remedy to feature scattering.

8 Discovering and Managing Variability among Berkeley DB Product Variants

We have shown how the problems of discovering and managing commonality/variability (see RQ1-5 in Section 1.1) are resolved in Chapter 3 - 7. For each of these problems, we evaluated the corresponding solution separately on different systems to make sure the solution is generally applicable.

In this chapter we conduct an overall evaluation of our approach to reengineering legacy software product variants into Software Product Line. The techniques we used in Chapter 3 - 7 are adopted to identify and manage the commonality, and variability among the product variants of Berkeley DB (BDB). Based on this, we can make claim that the solutions mentioned in Chapter 3 - 7 can integrate seamlessly to serve as a complete and systematic approach to reengineering legacy software product variants into Software Product Line.

8.1 Generating Input (Product Variants) for the Overall Approach

To conduct a controlled experiment on the application of our overall approach, a workbench which server as SPL to configure and generate the product variants is required. In this section, we introduce the target system (of the product variants) on which we evaluate our overall approach. We also discuss the workbench which has the ground truth on the knowledge of the variability and commonality of the target system. Finally, we introduce the generation of several product variants from the workbench.

8.1.1 The target system

The target system we choose for our case study should satisfy the following requirements:

• It should be at least medium size, which can prove that our overall approach is scalable for real industrial software products.

• A well-known system is preferred, which can be easy to be understood and compared with similar work by other research groups.

• The most important is that this system should be migrated into an SPL, and we have the knowledge of the variability and commonality in requirements and implementation. Furthermore, the variability techniques or SPL architecture should be deployed by the SPL to manage the variability.

According to the above requirements, the WFMS we used for case study in Chapter 3, 6 and 7 is not suitable. It is not public available, and the documentation on traceability between feature and code implementation for the existing product variants is not clear. Finally, we choose the system Berkeley DB (BDB) Java Edition [189] for the case study of our overall approach.

With a history of around 25 years, BDB was initially a part of the transition (1986 to 1994) from 4.3 Berkeley Software Distribution (BSD, sometimes called Berkeley Unix) to 4.4BSD. Now it has evolved into a software library that provides a high-performance embedded database for key/value data.

Berkeley DB Java Edition is an open source database engine, entirely implemented in Java. It can work as a standalone database (run as .jar file), or be embedded as a third party library in the Java application. Instead of being a relational engine, it provides the embedded storage, with open interfaces designed for programmers, not for DBAs.

The size of BDB Java is 84 KLOC, and the family of different BDB editions has been studied in [5,88,89,91,92,93,94,110,156]. Originally BDB Java was a single application, but Kästner and his colleagues have reengineered it as a Software Product Line (SPL) by adopting proper variability techniques such as AspectJ [96], CIDE [191] or FEATUREHOUSE [194]. Thus, the BDB Java satisfies the three requirements mentioned above, and we eval-

uate the overall approach on several product variants of BDB Java system generated by the variability techniques applied on this BDB-SPL.

SPL : [Logging] ConcurrTrans Persistance [**Statistics**] BTree Ops [**Memory_Budget**] :: generatedSPL ;

Logging : [**Logging_Finer**] [**Logging_Config**] [**Logging_Severe**] [**Logging_Evictor**] [**Logging_Cleaner**] [**Logging_Recovery**] [**Logging_DbLogHandler**] [**Logging_ConsoleHandler**] [**Logging_Info**] Logging_Base [**Logging_FileHandler**] [**Logging_Fine**] [**Logging_Finest**] :: _Logging ;

ConcurrTrans : [**Latches**] [**Transactions**] [**CheckLeaks**] [**FSync**] :: _ConcurrTrans ;

Persistance : [**Checksum**] IIO [**Environment_Locking**] Checkpointer [**DiskFullErro**] [**FileHandleCache**] IICleaner :: _Persistance ;

IIO : [**SynchronizedIO**] **IO** :: OldIO
| NIOAccess [**DirectNIO**] :: NewIO ;

NIOAccess : **ChunkedNIO**
| **NIO** ;

Checkpointer : [**CP_Bytes**] [**CP_Time**] [**Checkpointer_Daemon**] :: _Checkpointer ;

IICleaner : [**CleanerDaemon**] Cleaner [**LookAHEADCache**] :: _IICleaner ;

BTree : [**INCompressor**] [IEvictor] [**Verifier**] :: _BTree ;

IEvictor : [**Critical_Eviction**] [**EvictorDaemon**] **Evictor** :: _IEvictor ;

Ops : [**DeleteOp**] [**RenameOp**] [**TruncateOp**] :: _Ops ;

%% //Semantic Dependencies

Evictor or EvictorDaemon or LookAHEADCache implies Memory_Budget;
Critical_Eviction implies INCompressor;
CP_Bytes implies CP_Time;
DeleteOp implies Evictor and INCompressor and Memory_Budget;
Memory_Budget implies Evictor and Latches;
TruncateOp implies DeleteOp;
Verifier implies INCompressor;

Figure 8.1. The grammar of feature diagram of BDB Java

[89] shows the feature diagram of BDB Java. In this dissertation, we show its grammar in Figure 8.1 (see Section 2.1.2 about the grammar of feature diagram), as the grammar is more formal and clearer than the feature diagram. We highlight the variant features in the **bold** font in Figure 8.1, from which we can see BDB Java has around 50 features, inclusive of around 40 optional or alternative features.

Some important information about feature particulars is listed as follows:

- Number of features affecting over 10 class: 9

- Number of features affecting over 5 class (5 is exclusive): 7 + 9
- Number of features with mixed-grained impact: about 30

Note that: mixed-grained impact means the duality of the fine grained impact (at statement or even expression level) and the coarse grained impact (at method level or even class level). The interested readers can refer to [89,90] for more about feature particulars.

8.1.2 The usage of CIDE

CIDE (Colored Integrated Development Environment) is a software product line tool to manage the variability inside the core assets. It is annotative approach to compose/decompose the feature code [191] (also see Section 7.5). Nevertheless, it follows the paradigm of **virtual separation of concerns**, which avoids the **#if-def** style annotation or the physical modularity of the feature code. Unlike using the **#if-def** block to enable or disable the feature code, in Figure 8.2 CIDE uses color to highlight the feature code and enable/disable it on the AST according to the selection of the corresponding feature.

CIDE also allows the users to generate the product variant by selecting the desired features and deselecting the unwanted features via its UI interface of product generation. For the selected features, CIDE applies the constrain checking to assure the generated product is valid.

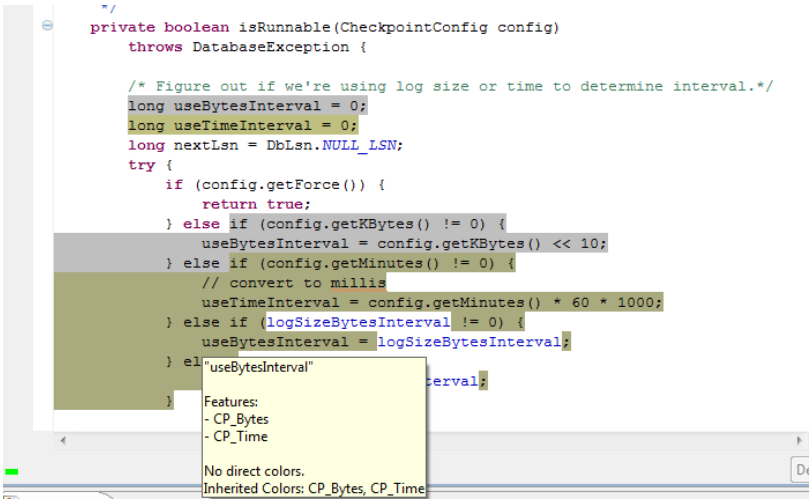

Figure 8.2. Feature code highlighting in CIDE

CIDE provides the knowledge of the feature to code traceability in the hovered box in its *ColoredJaveEditor* in Figure 8.2. It also has two kinds of views, *views on a feature* and *views on a variant* for ease to trace the feature among several products or a product with several features [88].

Although CIDE has the above merits in easy usage and complete knowledge of feature to code traceability, the major reason we prefer CIDE than other variability tools like AspectJ and FEATUREHOUSE is that the source code managed by CIDE is most close to the original source code. To get the physical modularity for the variant features (especially those fine-grained features), AspectJ has to adopt the Aspect's wormhole pattern [106], and FEATUREHOUSE has the similar placeholder (hook method) inside a method to insert the fine-grained feature's code via calling the hook method [5].

8.1.3 Randomly generated product variants

To evaluate our overall approach, a set of similar but different product variants are needed. With the capability of variability management provided by CIDE, we can customize the feature selection for any new generated product. We adopt the random strategy for the selection of variant features. Due to the large percentage of invalid and meaningless

feature combination, we randomly generate over 40 products, and choose several representative ones. The random generation strategy follows the heuristic rules below:

- The different representative ones should have the different major features. The major features include feature *Transactions*, *Statistics*, [*Evictor*, *Latches*, *Memory_Budget*] (we use "[]" to denote the mutual dependency among features), *DeleteOp*, *Incompressor* and *Verifier*.
- The feature dependency listed in Figure 8.1 (or in [89]) must be satisfied for each of the generated variants.
- The minor features should be uniformly distributed in these representative products, and any of the variant features should appear at least once in at least one product.

Table 8.1. The feature table with renamed features for product variants

Feature	Product 1	Product 2	Product 3	Product 4	Product 5
Base code	√	√	√	√	√
Logging_Finer	√		√	√	
Logging_Config		√ (Logging_Customize)	√		
Logging_Severe		√	√		
Logging_Evictor	√ (Logging_Expel)			√	
Logging_Cleaner		√ (Logging_Sweeper)		√	√
Logging_Recovery		√	√		√
Logging_DbLogHandler					√
Logging_ConsoleHandler		√		√	√
Logging_Info	√	√		√	
Logging_FileHandler	√				√
Logging_Fine			√	√	
Logging_Finest	√	√		√	
Latches	√ (Locks)		√	√	√
Transactions	√	√	√ (Processing)		√
CheckLeaks	√	√		√	
FSync	√		√		
Checksum		√		√	√ (CheckTotality)
SynchronizedIO	√				
IO	√			√	

ChunkedNIO			√		
NIO		√			√ (NewIO)
DirectNIO			√		√
Environment_Locking	√			√ (Circumstanc stanc-es_Locking)	
CP_Bytes	√			√	
CP_Time	√		√	√	
Checkpointer_Daemon		√	√	√	√
DiskFullErro			√	√	√
FileHandleCache	√		√		
CleanerDaemon			√		√
LookAheadCache				√ (LookFor-wardCache)	√
Statistics		√	√	√	√ (Datum)
INCompressor		√	√	√	
Verifier		√ (Testify)		√	
Critical_Eviction			√		
EvictorDaemon				√	√
Evictor	√ (Expel)		√	√	√
DeleteOp			√ (ClearOp)	√	
RenameOp	√	√			√
TruncateOp				√ (DropOp)	
Memory_Budget	√		√	√	√
# Total Features	19	17	22	26	20

After the random generation of over 40 product variants, we still need to pick up several representative product variants for evaluation as the 40 products are too many and with unnecessary repetition. To simulate the real context of a software family, we filter those products with very little variant features or almost all variant features. Although the BDB Java allows the thousands of valid configurations of feature selection, in our preliminary evaluation of the overall approach we plan to use 5 product variants (we call them *P1*, *P2*, *P3*, *P4* and *P5* in Table 8.1). As WFMS maintains 5 or 6 major versions of its product to reduce the cost [182], a minimum of 5 representative products could be the starting point for us to analyze and manage the variability among them.

We listed the feature table of the 5 product variants for the evaluation in Table 8.1. For product 1, it has no major feature *Statistics*, *DeleteOp*, *Incompressor* and *Verifier*. For

product 2, it has no major feature [*Evictor*, *Latches*, *Memory_Budget*] and *DeleteOp*. For product 3, it has no major feature *Verifier*, and this product is comparatively the most comprehensive one. For product 4, it has no major feature *Transaction*. Finally, the product 5 has no major feature *DeleteOp*, *Incompressor* and *Verifier*.

For the tree-like feature structure, namely Product Feature Models (PFMs) of these five product variants, we also applied the random renaming as we did in Section 3.4.2.1. As shown in Table 8.1, after renamed, the new feature's name is in the bracket. We use the WordNet [50] to find the synonym for renaming, by which we can show that the feature matching among product variants does not rely on string matching.

8.2 Analyzing Variability among Product Variants by the Sandwich Approach

In this section, we adopt the techniques introduced in Chapter 3 to find the variability among the requirements of the product variants generated in previous section 8.1. We also apply the clone differencing techniques in Chapter 4 to recover the variability among the implementation of these product variants. Finally, we map variability at the requirement level to the variability at the implementation level by using techniques in Chapter 5. By the above steps, we can have a comprehensive view on how the product variants are different from each other in requirements and implementation.

8.2.1 Understanding requirement variability in BDB-Java product variants

We adopt the technique used in Section 3.3 to compare the PFMs of these five product variants. Briefly, we compare these PFMs based mainly on the structural information and slightly on the lexical similarity of feature name. As we have the ground truth of features contained in each product in Table 8.1, we can easily compare our reported results with the actual results in Table 8.2.

Table 8.2. The actual and expected results of PFMs comparison for BDB-Java product variants

Left vs. Right	The Actual Results			The Expected Results		
	#Valid Overlapped Features	# Valid Left Unique Features	# Valid Right Unique Features	#Overlapped Features	#Left Unique Features	#Right Unique Features
P1 vs. P2	4	11	10	6	13	11
P1 vs. P3	7	8	11	9	10	13
P1 vs. P4	12	5	12	13	6	13
P1 vs. P5	4	10	11	7	12	13
P2 vs. P3	5	7	12	8	9	14
P2 vs. P4	10	6	13	11	6	15
P2 vs. P5	10	5	8	10	7	10
P3 vs. P4	11	9	13	12	10	14
P3 vs. P5	10	9	7	11	11	9
P4 vs. P5	11	13	7	12	14	8

In Table 8.2 which lists the 10 times of pair-wise comparison for 5 products, under the column "The Expected Results", the column "#Overlapped Features" refers to the number of the features contained in both products, "#Left Unique Features" means the number of the features only exists in the left product. Similarly, "#Right Unique Features" means the number of the features only exists in the right product. Under the column "The Actual Results" we show the valid results from the actually reported candidate results. Simpler than the result type of the tuple $< f_1, f_2, Rename >$ namely $Rename$ used in Section 3.3.4 (see more result types like *Split* and *Merge*), for these 5 product we only define three result types $< f_1, f_2, M >$, $< f_1, null, LUM >$ and $< null, f_2, RUM >$. The reason is that there is not complicated evolution among these products such like splitting and merging features. Thus, we define type M for matched features, LUM for the left unmatched feature (or the left unique feature) and RUM for the right unmatched feature (or the right unique feature).

As shown in Table 8.2, the overall accuracy of the reported results is good. Averagely, we can recover over 80% correct results. Furthermore, there are some interesting observations as follows:

Observation 1: if two products share more overlapped features, usually the results may be better. For example, product 1 and 4 share about 50%-66% common features, and the

results have only one missing $< Locks, Latches, M >$ (see Table 8.1), one missing $< FileHandleCache,\ null, LUM >$ and one missing $< null, DiskFullErro, RUM >$.

The reason is that the same feature *Locks/Latches* have different structural or neighbour information and the lexical similarity is not high enough. Thus, our algorithm fails to match them. But our algorithm indeed considers the actually different features *Disk-FullErro* and *FileHandleCache* the same, since these two features have very alike matched neighbour structure. Actually, the same feature of *Environment_Locking* in product 1 and *Circumstances_Locking* in product 4 is matched due to the above reason. The intuition behind this observation is that more overlapped common features between two products usually lead to the more similar structure between their PFMs, which further help in our matching algorithm relying mainly on structural information.

Observation 2: if two products share not many overlapped features, usually the results may be still acceptable. For example, product 2 and 3 share about 33% common features. Still, the results miss $< Logging\_Customize, Logging\_Config, M >$, $< Transaction, Processing, M >$ and $< Checkpointer\_Daemon, Checkpointer\_Daemon, M >$. The first two of the above three missing pairs are due to renaming. And the last missing pair is because of the unmatched structure of feature *Checkpointer_Daemon* and its neighbors. The results also have 2 left unmatched features and 2 other right unmatched features. The reason is that our algorithm mistakenly matches <*NIO*, *ChunkedIO*> as a pair, and <*RenameOp, ClearOp*> as a pair. But actually they are four distinct features. The explanation for this observation is that without structural similarity the comparison can only rely on lexical similarity. Since there is no intensive renaming among products, the lexical similarity still achieves the acceptable accuracy for the result.

Note that feature *Logging_Finest* in product 2 will not be matched with feature *Logging_Fine* or *Logging_Finer* in product 3, although their names are similar and have the same parent feature. Actually, we apply some checking between the comparisons. In Table 8.1 product 1 has *Logging_Finer* and *Logging_Finest* at the same time, and product 4 has *Logging_Fine* and *Logging_Finest* at the same time. Thus, we can infer that these three features are distinct features, not renamed features.

Note that the above results are from direct comparison of tree-like PFMs. If we take the feature descriptions into accounts, the results may even be improved to a completely correct level.

8.2.2 Understanding implementation variability in BDB-Java product variants

To understand the variability at the implementation level, we adopt the cloned detection and software differencing techniques. The cloned detection technique is used to find the coarse-grained feature code, while the differencing technique we introduced in Section 4.4 is mainly used to find the fine-grained feature code.

The coarse-grained feature code means the feature relevant code that is at the file or method level. If a class/method is not reported as the clone class/method between two product variants, it is usually because this class/method is related to a feature only supported by one product variant. In the scenario of comparing variants *P1* and *P2*, the class *FSyncManager* in *P1* is not reported as a clone class with any class in *P2*. *FSyncManager* is the relevant code of feature *FSync*, which is supported by *P1* and not by *P2*. Except those getter and setter methods, none of methods in *FSyncManager* is reported as a clone method with any method in *P2*.

The fine-grained feature code means the feature relevant code that is at the code block, statement (and branch) or expression level. Sometimes, the impact of a fine-grained feature may only exist at the statement or expression level. For example, the feature *Log-*

ging_ConsoleHandler supported by *P5* only has the impact at the statement and branch level. As this logging feature is a crosscutting concern to many functions, this feature's code is scattered in different methods. Since one missing statement or branch does not change the result of clone detection (it may create gapped clones, which are also clones, see Section 2.2.1), feature *Logging_ConsoleHandler* 's code are scattered in clones. By comparing these instances of gapped clones, we can identify the feature code.

Actually, the differences between two product variants come from two parts:

• The variant features that are only supported by one of the two compared product variants. For any two compared product variants, actually we have found the differences at the feature level in In Table 8.2, from which we can know the information like F_{P1-P2}, the set of features that are in *P1* not in *P2* (*P1-P2*). Thus, the relevant feature code of features in F_{P1-P2} are in I_{P1-P2}, the set of code fragments that are in *P1* not in *P2* (*P1-P2*).

• The variant features that are supported by both product variants. Even if a feature is supported by two product variants, more often than not it may still have the different feature code. The reason is because of the feature interaction or feature dependency. For example, *P1* and *P3* both have feature *Latches*, but they have the different feature code of *Latches* due to feature interaction. As *P3* supports feature *Statistics* and *P1* does not, the code relevant to both *Latches* and *Statistics* is supported in *P3*, not in *P1*.

Note that the differences that come from the above two parts may be at any granularity level. In Table 8.3, we show the implementation differences between any pairs of these 5 product variants.

The results show that the differences are mainly the fine-grained feature code. This phenomenon is consistent with the previous literatures [90,161]. The BDB Java has many fine-grained features [90], and this can be generalized to the point that many concerns (features)

are usually tangled and scattered across the system [161]. That is the reason why orthogonal separation concerns to design all features as coarse-grained is not feasible.

Another important observation is that a small size of a feature difference set may not lead to a small size of an implementation difference set. For example, the feature difference set F_{P5-P2} (see *#Right Unique Features* at *P2 vs. P5* in Table 8.2) has only 10 variants inside, but it contains over 100 additional methods, around 1000 addition statements and 100 additional branches in I_{P5-P2}. We manually checked and found that around half of these implementation differences are due to a major feature *Latches*. Similarly, a large size of feature difference set like F_{P4-P3} (see *#Right Unique Features* at *P3 vs. P4* in Table 8.2) does not necessarily have a large size of implementation difference set. It is because none of the 14 features in F_{P4-P3} is a major feature.

Table 8.3. The implementation differences among product pairs

	#Additional Classes	# Additional Methods	# Additional Blocks	#Additional Statements	#Additional Branches	#Additional Expressions
P1 – P2	17	106	47	1036	106	23
P2 – P1	18	82	20	429	45	47
P1 – P3	0	15	12	137	9	1
P3 – P1	13	98	23	480	50	54
P1 – P4	8	53	29	172	17	56
P4 – P1	20	113	21	475	46	57
P1 – P5	2	29	16	179	10	6
P5 – P1	15	81	16	383	32	53
P2 – P3	5	25	6	140	12	2
P3 – P2	18	132	44	1090	114	31
P2 – P4	7	43	32	181	21	55
P4 – P2	19	127	51	1091	111	32
P2 – P5	5	32	10	171	18	0
P5 – P2	18	108	37	982	101	23
P3 – P4	9	59	34	246	30	58
P4 – P3	8	36	15	206	18	6
P3 – P5	3	57	17	231	28	13
P5 – P3	3	26	6	92	9	7
P4 – P5	7	68	21	276	23	12
P5 – P4	8	60	29	177	16	58

8.2.3 Understanding implementation variability in BDB-Java product variants

Consider a simplified illustrative example shown in **Figure 8.3**(a). The product variant P_A supports feature f_1, f_2 and f_3, implemented by code units I_1, I_2 and I_3 respectively. P_B supports f_1, f_2' (renamed or modified of f_2) and f_4, implemented by I_1, I_2' and I_4 respective-

ly. P_C supports f_1, f_3' and f_5, implemented by I_1, I_3' and I_5 respectively. f_1 exemplifies basic features that are commonly available in product variants, such as core algorithms or utility functions for implementing other advanced features.

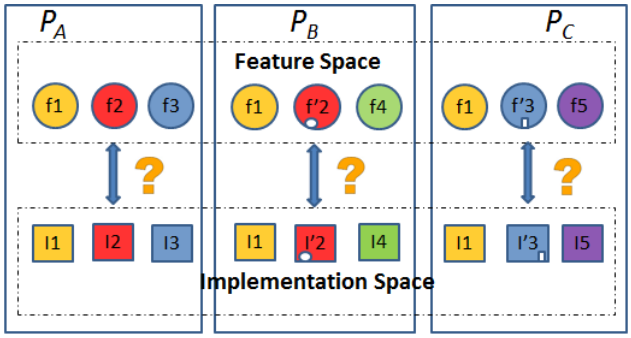

Figure 8.3(a). Feature location in product variants

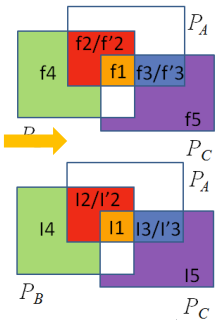

Figure 8.3(b) Results of differencing and FCA analysis

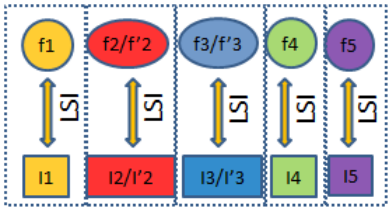

Figure 8.3(c). Features and implementaiton partitions for IR

Our approach introduced in Section 5.3 effectively incorporates Formal Concept Analysis (FCA), and IR techniques. As we found the feature and implementation differences among these variants, Figure 8.3 (b) shows the results of applying FCA to the three product variants shown in Figure 8.3(a). This essentially generates five partitions of features and their corresponding implementations as shown in Figure 8.3(c). Finally, given a feature partition and the corresponding code-unit partition, Latent Sematic Indexing [41] (LSI) is used to identify code units that implement a specific feature [118, 119, 134].

For the feature differences we identify in Section 8.2.1, after we consider the feature descriptions, we can manually validate the differences to get completely correct results. From these trustable feature differences, we can apply FCA to them to separate single feature, which makes the usage of information retrieval as an optional step. For the FCA on BDB Java, 41 variant features are the attributes, 20 difference sets are the objects. Finally, FCA will generate 136 concepts.

As can be seen in Figure 8.4, we may mainly concern with the 23 concepts that own attributes (features). 11 out 41 features can be ideally separated into a partition with only one feature. For example, for feature *Verifier*, which exists in *P2* and *P4*, we get the intersection of I_{P4-P5}, I_{P4-P3}, I_{P2-P5}, I_{P2-P3}, I_{P4-P1}, I_{P2-P1} in Figure 8.5. Intuitively, the above intersection means that the code of feature *Verifier* should exist in *P4, not P5 or P3 or P1*, and *P2, not P5 or P3 or P1*. Thus, by intersecting these code difference sets, we can separate feature *Verifier* into a minimal equivalent partition (see Section 5.3.4). Then considering the acceptable size of this partition, it is not laborious to identify which code fragments belong to feature *Verifier* and which code fragments are feature interactions existing in *P2* and *P4*.

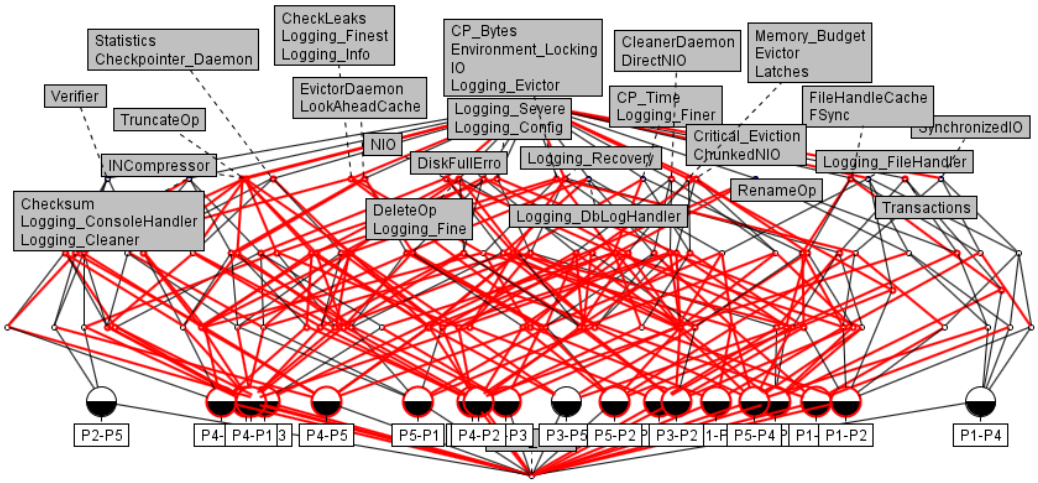

Figure 8.4. The FCA for the features of 5 product variants

There are also 8 concepts (partitions) with 2 attributes (features). Actually, in this BDB Java case study, there is no concept (partition) with many features inside like that in the Linux product family (see Section 5.5.3). Therefore, it is not necessary to apply information retrieval technique to the feature and the corresponding partition, as the code corpus of the partition is not large enough for information retrieval to perform well.

The drawbacks of our sandwich variability-recovery approach are that the feature dependency and interaction are still hard to identify from the implementation level. First the product variants may not cover all the code affected by feature interactions. Besides, our current approach to find the feature interaction is still in an ad-hoc way.

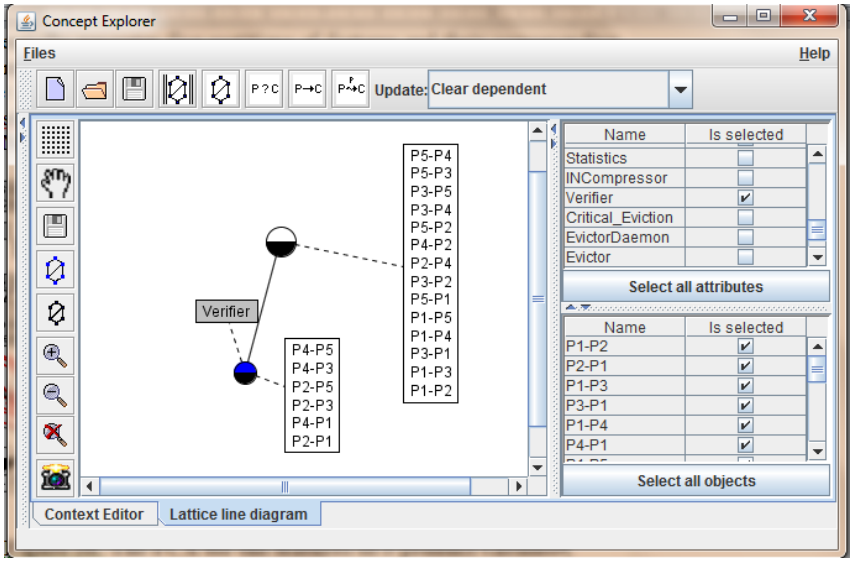

Figure 8.5. Separation of single feature by intersecting code difference sets in Concept Explorer

8.3 Managing Variability in B-DB by XVCL

After we identify those variant features in requirements as well as their corresponding code, it is desired to design reusable, customizable components and plug in the variant features as the users wish. In this section, we first briefly compare XVCL with preprocessing.

Then we explore the application of XVCL as a uniform variability technique for the code differences of BDB Java variants found in Section 8.2.

8.3.1 Preprocessing as a Variation Mechanism

It is useful to categorize features that differentiate product variants as follows: *Fine-grained features* affect many base components, at many variation points; *coarse-grained features* can be contained in base components that are included into a custom product that needs such features; *mixed-grained features* involve both fine- and coarse-grained impact.

Coarse-grained features can be easily accommodated into product variants with **#include** directives placed in base components, or using a build tool such as *make*. Fine-grained feature code is kept together with the base code under conditional compilation directives (e.g., #ifdef). Each variation point in the base code (i.e., point affected by some features) corresponds – in more or less explicit way - to a specific combination of features that affect that point.

In our model preprocessing notation, <**select-option**> commands mark variation points. Suppose features A, B, C and D can be optionally included into product variants, members of some software Product Line. Suppose further that **Base_X** and **Base_Y** are two reusable base components (source files) for that Product Line. Feature A interacts with **Base_X** as shown in Figure 8.6, **Base_X** contains <**select**> command with <**option** A> that marks that variation point. This is a simple case of a feature affecting the base code without interactions with other features. Such cases are also easily handled by preprocessing directives **#ifdef**.

The above solution is put to work by means of parameters and expressions. Each feature is represented by a parameter. For example, parameter *a* represents feature A, *b* – feature B, and so on. The top-most SPeCification file, **SPC,** <**set**>s values of parameters and in that way specifies which features we need in a product variant. In Figure 8.6, we wish to

select features A and D, so parameter *a* is <**set**> to "A" and *d* is <**set**> to "D". Parameters for unwanted features are <**set**> to null string " ". Notation **@v** means a reference to variable *v*.

To derive a product variant that implements selected features, base components are processed starting with **SPC**, in depth-first order via <**adapt**> links. During this traversal, the Processor interprets commands and emits code contained in base components to the output files, just as any preprocessor does. <**adapt**> is analogous to cpp's **#include**.

During processing, values of parameters propagate down to the <**adapt**>ed base components, and are used to identify <**option**>s relevant to selected features. Features that affect a base component at a given variation point are identified by parameter *v* that controls <**select**>. In SPC of Figure 8.6, we selected features A and D, so in **Base_X**, parameter *v* is <**set**> to "A", and code under <**option** A> is included into the product variant.

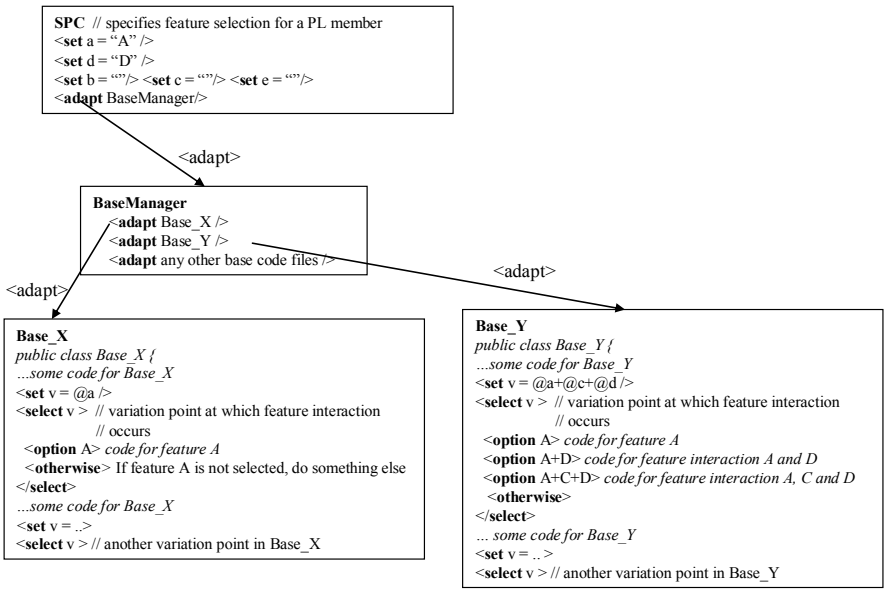

Figure 8.6. Managing fine-grained features in base components with preprocessor

Base_Y includes a variation point with interacting features A, C and D. As we selected features A and D, parameter *v* is <**set**> to "A+D" in **Base_Y**, where the value of *v* is defined as concatenation (operator '+') of values of parameters *a*, *c*, and *d*.

Effectively, each variation point is clearly marked with names of features that affect that point, and interact at that point. All the variation points that are associated with a certain feature are inter-linked by means of parameters with global scope.

Our <**select**> is analogous to the **#if** .. **#elif** .. **#else** .. **#endif** directive. It is functionally equivalent to **#ifdef**, but avoids problems of nested **#if**'s analyzed in [155]. <**set**> command is analogous to the #define directive except that values propagate globally across base components. In cpp, parameters set at the command line also propagate to all the preprocessed files. Expressions at <**option**>s may not have direct counterpart in some preprocessing systems, but this is a minor extension, and can be easily implemented to any preprocessor.

As not all the combinations of features can be legally selected for any given custom product, it is a good practice to validate a given feature selection before the customization process starts. Such validation can be done using formal methods Z, Alloy and OWL DL [159,167].

8.3.2 Preprocessing problems in Berkeley DB

We use Berkeley DB to illustrate common preprocessing problems in handling product variants. Many features can be optionally included into custom DB systems. In that sense, Berkeley DB forms a Product Line, whose members implement different selections of features. Berkeley DB designers chose to use runtime mechanisms to accommodate required features into a custom DB system. In earlier studies, Berkeley DB was converted into a Product Line in which features were managed at the construction time (i.e., before execu-

tion) with AspectJ [89] and CIDE [90]. In our study, we also managed features at the design-time with a preprocessing notation described in Section 8.3.1.

8.3.2.1 Overview of our study

The Berkeley DB consists of five subsystems, namely *access methods* to create and access the database, *B^+-tree* to store data as key/value pairs, *caching and buffering* to increase database performance, *concurrency and transaction* to handle concurrent and rollback facility, and a *persistence layer*. These five subsystems are designed with 232 base components that form an architecture shared by all the DB system variants. Two of those files, namely **FileProcessor**, and **LogBuffer** are shown in Figure 8.7. In our experiment, we did not change the original design of the Berkeley DB.

38 features such as `IO`, `LookAheadCache` or `DiskFullHandler` can be optionally included into custom DB system variants. Berkeley DB designers used runtime mechanisms to configure features into custom DB products. For example, settings for `LookAheadCache` feature are stored in an environmental property file (called *je.properties*), and developers can change this file to include and re-configure the behavior of this feature.

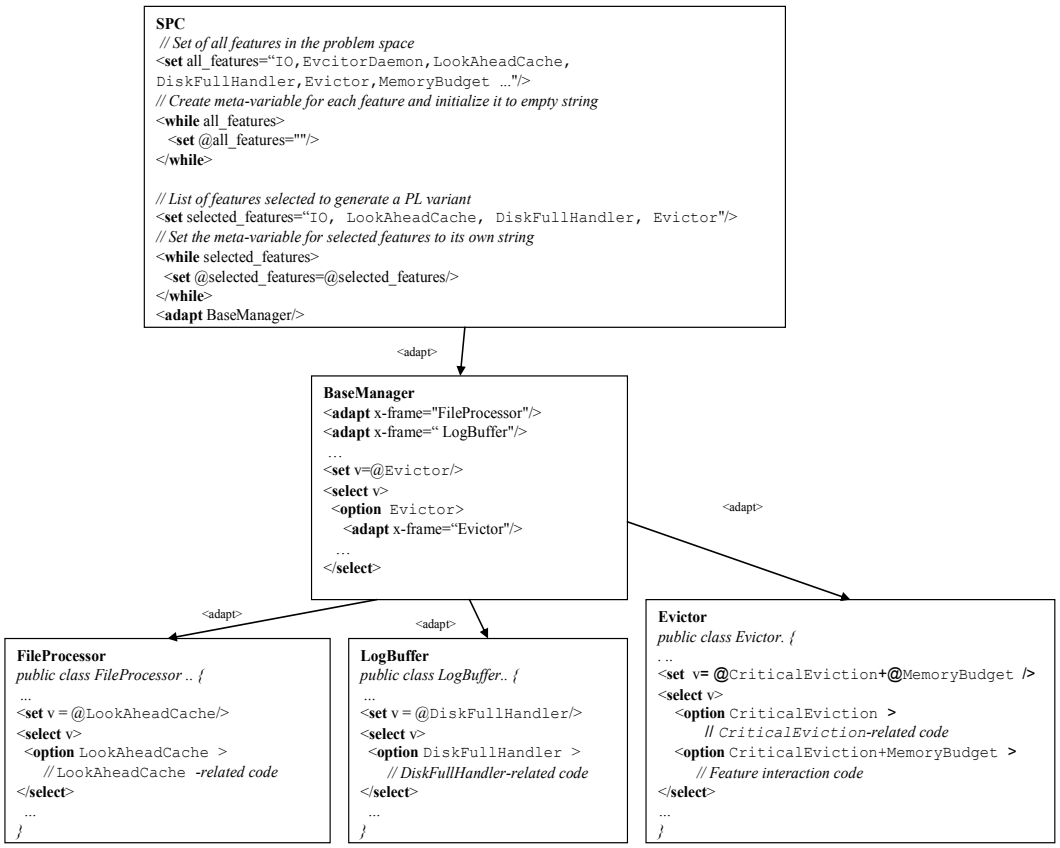

Figure 8.7 A preprocessing solution to managing features in Berkeley DB

In our study, we addressed 22 Berkeley DB features ranging from simple to complex. First, we analyzed the semantics of the features we decided to work with, and the way they affected the base DB code. We converted original runtime strategies for feature management to equivalent preprocessing strategies, in a similar way as Kastner et al. converted them to AOP. For that, we instrumented each base component affected by features with preprocessing commands to manage feature impact on the base code.

Our preprocessing representation for Berkeley DB, shown in Figure 8.7, follows conventions described in Section 8.3.1. Parameter *features_selected* in **SPC** specifies features to be included in a required DB variant, in case of our example, features `IO`, `LookAheadCache` and `DiskFullHandler`. **SPC** adapts **BaseManager**, which in turn

adapts all DB base components. **BaseManager** plays the role of an integrator/composer of a custom DB variant in terms of its base components, propagating customizations for selected features to the base components. For clarity, in Figure 8.7, we have shown only two of such base components, namely **FileProcessor** and **LogBuffer**.

In addition, **BaseManager** also <**adapt**>s feature-specific base components. This is because for some of the features, besides adding small fragments of code at variation points in base components upon feature selection, we must also add new files to the DB base. Class *Evictor* in Figure 8.7 exemplifies this situation, and so does feature `Statistics` that adds classes StatsConfig and BtreeStats to the DB base (not shown in Figure 8.7.)

8.3.2.2 Patterns of feature impact on DB base components

Features that affect small number of base components at small number of variation points, and without interacting with other features are easily handled by preprocessing. Such features may require adding new classes to the DB base, e.g., features `Evictor` and `Statistics`, or adding member variables and methods to classes. Each such feature impact is handled by placing these member variables and methods within <**select-option**> block in affected base components.

Among 22 features in DB that we addressed in the experiment, five were preprocessing-friendly.

First type of complication occurs when the number of variation points at which features affect a given base component grows. The impact may come from one or more features. Then, the base code becomes densely populated with <**select**>s, which is one of the often mentioned drawbacks of using preprocessing to handle product variants. Among other examples, 12 features affect base component **Environmentimpl** inducing 38 variation points, and 10 feature affect file **FileManager**, inducing 40 variation points. The number of variation points per class ranges from 1 to 35, with average 5.72.

Further complications appear when the impact of one feature becomes scattered over many variation points, and spreads through many base components. For example, feature `MemoryBudget` affects 32 DB base components at a total of 190 variation points, `Statistics` (34 variation points), `CheckSum` and `Evictor` (each with 27 variation points), and `CriticalEviction` (23 variation points). When modifying scattered features, we must propagate changes to all the relevant variation points. Maintenance becomes difficult and error-prone.

Feature dependencies and interactions bring new dimension of difficulties for preprocessing. A feature f_1 depends on feature f_2 if f_1 refers to code of *f2*, or if the presence or absence of *f2* in a product variant affects implementation of *f1*.

Feature dependencies cause feature interactions. In preprocessing solution, feature interactions show as variation points with <**option**>s labeled with names of interacting features, or - in more complex interaction situations - as multiple <**option**>s under a <**select**> command.

Figure 8.8 shows an example of feature interactions in which feature `CriticalEviction` interacts with feature `Evictor`, and feature `MemoryBudget` interacts with feature `CriticalEviction`. The net result of those interactions shows at variation points in file **Evictor**. If we select feature `CriticalEviction`, method *doCriticalEviction* must be included into the **Evictor**. If at the same time we select feature `MemoryBudget`, method *doCriticalEviction* must be modified with code related to that feature.

Feature code scattering and feature interactions lead to hidden dependencies among many variation points. These dependencies have to be understood to modify code without unwanted side-effects. Figure 8.8 shows an example of a hidden dependency induced by feature interactions. Features `CriticalEviction` and `MemoryBudget` affect the **Evictor**, which is only included if the `Evictor` feature is selected. Therefore, once the

variation point that includes the **Evictor** is removed, all variation points introduced by the features `CriticalEviction` and `MemoryBudget` in file **Evictor** will be automatically affected.

```
Evictor
public class Evictor {
…
<set v= @CriticalEviction+@MemoryBudget />

<select v>
   <option CriticalEviction>
    public void doCriticalEviction ( )
       throws DatabaseException {
                  doEvict(SOURCE_CRITICAL, true);
         }
  <option CriticalEviction+MemoryBudget>
public void doCriticalEviction ( )
      throws DatabaseException {
        MemoryBudget mb = envImpl.getMemoryBudget();
            long currentUsage  = mb.getCacheMemoryUsage();
           long maxMem = mb.getCacheBudget();
        ...
}
</select>
...
```

Figure 8.8. Feature interactions

8.4 Summary

In this chapter, we evaluate our sandwich variability- recovery approach for a set of similar products of Berkeley DB Java Edition. Although these product variants are artificially randomly generated from the original all-in-one version provide by CIDE [191], we still follow some heuristic rules to make the product variants more real.

Given these product variants, we build Product Feature Models (PFMs) and compare them pair-wisely. The results showed that our PFMs comparison algorithm can achieve a reasonably good accuracy. If we consider the feature descriptions, we can get almost completely correct results.

After we compare the PFMs from requirements, we also compare the source code of these product variants. We apply the clone detection and software differencing techniques to find the fine-grained and coarse-grained feature-relevant code. We use clone detection to filter out the non-cloned classes or methods as additional classes or methods (coarse-

grained feature-relevant code) among variants, and use CloneDifferentiator to identify the PDGs' differences as additional blocks, statements, branches or expressions methods (fine-grained feature-relevant code). The comparison is also done pair-wisely.

To bridge the feature differences with the code differences, we use the FCA and IR techniques. The FCA helps generate the disjoint, minimal equivalent partitions of the features and code units. From the feature partitions, we know which features are owned by which intersections of scenarios. Here, the scenario means a difference set of a comparison of two product variants. According to the intersection of scenarios, we can get the results of the intersection of code units. The results of the intersection of code units contain the corresponding feature relevant code. The results of FAC for BDB Java product variants show that most features are contained in small partitions that have only 1 or 2 features. The above ideal separation of features makes application of IR as an optional step.

After we have identified the commonality and variability, pre-processing directives can be viewed as kind of a variability technique that extends conventional programs by adding configuration knowledge to them. The XVCL subset – XCpp, which is a meta-data, uses queries to analyse the configuration information. XCpp approach can be useful add-in technique for other systems that work with meta-data: Pragmas in Smalltalk contain extra information that can be interpreted by tools. Java and JEE annotations [196] contain meta-data that extends program behaviour (for example, weaving AOP's advices can be expressed in some annotation systems). Annotations can be analysed for understanding in XCpp-like fashion.

XCpp described in this paper can be improved by providing more synthetic, visual presentations of features under analysis. Integrating XCpp feature analysis with analysis of the base/feature code would be beneficial, but such a system would substantially diverge

from pre-processors, making the solution language-dependent. Nevertheless, this is an interesting avenue to explore.

We presented XCpp system that helps one manage scattered feature code and feature interactions, improving readability of programs manipulated by pre-processors. XCpp applies queries to help developers navigate through feature-related code, showing features under analysis, while hiding other features. We analysed pre-processing problems and illustrated XCpp benefits in a study of a sizable Berkeley DB system.

9 Conclusion and Future Work

Software maintenance usually takes 70% of overall project costs. A common misconception about maintenance cost is that people thought maintenance merely includes bug fixing. However, one study indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions [132]. Actually, much of the maintenance efforts are due to adaptive or managerial maintenance – costs of modifying a software solution to allow it to remain effective in a changing business environment, or maintaining multiple versions for various customers in sync.

This dissertation aspires to contribute to this line of research by automatically reverse-engineering a family of software legacy systems into SPL to reduce the maintenance efforts. Specifically, we focus on resolving the problems of discovering and managing product variability respectively in two phases of engineering, namely domain engineering and application engineering. This dissertation can be regards as a compilation of the author's work on discovering variability at different levels, and managing variability by different variability techniques. The content and materials of the dissertation are also organized along this line: discovering the variability in requirements, discovering the variability in implementation, locating variant features (mapping variability in requirements to variability in implementation), evaluating various traditional variability techniques, and finally evaluating XVCL for variability management.

9.1 Summary of the Dissertation

Understanding of how features evolved in product variants is a prerequisite to transition from ad hoc to systematic SPL reuse. In Chapter 3, we propose a method that assists analysts in detecting changes to product features during evolution. We first entail that features and their inter-dependencies for each product variant are documented as product feature

model. We then apply model differencing algorithm to identify evolutionary changes that occurred to features of different product variants. We evaluate the effectiveness of our approach on a family of medium-size financial systems. We also investigate the scalability of our approach with synthetic data. The evaluation demonstrates that our approach yields good results and scales to large systems.

In Chapter 4, we present an automated approach to identify contextual differences of software clones, which contain the potential variant features. We represent clone contexts as program dependence graphs. We then apply graph differencing technique to identify seven types of elementary contextual differences that may affect the computation performed by clones. Based on these elementary contextual differences, developers can define queries to automatically distill clones relevant to a given maintenance task. We have implemented our approach in a tool called CloneDifferentiator. We evaluate CloneDifferentiator in two empirical studies aiming at refactoring Java IO library and Eclipse unit test suites.

In Chapter 5, we discuss problems that hinder direct application of IR techniques to identify feature-relevant code units in a collection of product variants. To counter these problems, we present an approach to support effective feature location in product variants. The novelty of our approach is that we exploit commonalities and differences of product variants by software differencing and FCA techniques so that IR technique can achieve satisfactory results for feature location in product variants. We have implemented our approach and conducted evaluation with a collection of nine Linux kernel product variants. Our evaluation shows that our approach always significantly outperforms a direct application of IR technique in the subject product variants.

In Chapter 6, we investigate the variability management in industrial product. Fudan Wingsoft Ltd. developed a product Line of Wingsoft Financial Management Systems

(WFMS-PL) providing web-based financial services for employees and students at universities in China. The company uses a wide range of variation mechanisms such as conditional compilation and configuration files to manage WFMS variant features. We study this existing product line and find that most variant features have fine-grained impact on product line components. Our study also shows that different variability techniques have different, often complementary, strengths and weaknesses, and their choice should be mainly driven by the granularity and scope of feature impact on product line components.

Chapter 7 follows up our earlier study of an SPL at Fudan Wingsoft Ltd that reveals potential scalability problems of multiple variability techniques. As a remedy to the above problems, in the follow-up study we replace multiple traditional variability techniques originally used in the Fudan Wingsoft product line, with a single, uniform variability technique of XML-based Variant Configuration Language (XVCL). This chapter provides a proof-of-concept that commonly used variation techniques can indeed be superseded by a subset of XVCL, in a simple and natural way. We describe the essence of the XVCL solution, and evaluate the benefits and trade-offs involved in multiple variability techniques solution and single variability technique - XVCL solution.

Chapter 8 integrates all the previously adopted techniques to conduct a preliminary evaluation of our overall approach to discover and manage variability inside a family of Berkeley DB Java products. We follow some heuristic rules to generate five major BDB Java product variants from CIDE [191], which cover all the variant features. For these variants, we apply the PFM comparison technique used in Chapter 3 to discover the variability in requirements. Then we apply the clone detection and clone differencing techniques used in Chapter 4 to discover the variability in implementation. To bridge these two levels of differences, we used FCA and IR techniques in Chapter 5 to facilitate locating variant features. Finally, we discuss about the pre-processing and XVCL, and evaluate XCpp

(XVCL-based Pre-processing) as variability technique to manage the variability in BDB Java product family. Overall, the results showed that our sandwich approach can help automatically and systematically identify the variability across product variants with reasonably good accuracy, and XCpp can help mitigate the problems of variability management with meta-data and query system.

9.2 Contributions and Perspective

The contribution of this dissertation is mainly twofold. There are many existing established studies on SPL practice [111], variability modeling [152] and variability techniques [28,29]. In this dissertation, our intention is not to invent new variability modeling methods, variability techniques, and software process towards SPL. We propose a systematic and automatic approach to reengineering software product variants into SPL. Thus, fundamentally the thesis contributes to systematic reuse of legacy products.

We also bridge the work of variability recovery with variability management. The knowledge we found in variability recovery like the granularity of the variant features can better provide guidelines in the variability management. We investigate the merits and drawbacks of various traditional variability techniques. As the different granularity of features matches the different variability techniques, the knowledge of granularity is important to the success of the application of variability techniques. Thus, another major contribution is information integration for the variability analysis and variability management.

In this dissertation we are doing much work from the perspective of reverse-engineering. Our study relies on the techniques such as model differencing, software clone detection, formal concept analysis, and information retrieval. All these techniques are usually applicable to mass of data, aiming to dig out useful information from the data. We are one of earliest groups who propose the integration of clone detection with model differencing, which compares the PSGs of clone instances to help understand clones. We are also

one of earliest groups who propose to locate variant features by considering the variability and commonality inside a family of product variants.

Early studies [8,20,107] indicated that 5%-10% of clones are a kind of homogeneous crosscutting concerns. And these concerns sometimes are the variant features for the system. The unmatched code units in clone detection may imply the potential existing of coarse-grained feature impact, and the differences among clone instances of two products may imply the existing of fine-grain feature impact. XVCL, which was mainly used for clone management or code reduction [72], is also applied in our study for the variability management. Thus, from this view, we compare XVCL with Pre-processing and a meta-level configuration tool as a variability technique in this dissertation.

9.3 Future Work Plan

The current concern of variability analysis in the dissertation is to identify the variant features' relevant requirements and code units. Our work still remains at the stage of comparing product variant in requirements and implementation. We have not figured out what knowledge can be further unveiled from the current matching results, and in what way.

In the following-up study, we may focus on the recovery of the feature dependencies and feature interactions. The feature dependencies refer to the features' inter-relationship in requirements, such as one feature *requires* or *excludes* another feature. Our current PFMs comparison only reports the matched and unmatched features, but fails to report what possible relationship exists among these matched features. The association rules mining techniques [1] or sub-tree mining techniques [31] may be helpful for this problem domain. From the matching result, we may further report that information such as which feature always appears together with some other feature(s), and which features never appear together.

For the current feature location in a set of product variants, we have not systematically investigated the impact of feature interaction. Feature interactions refer to tangling of two features' code units. Although the current results show that the accuracy is still acceptable, we hold the position that the feature interactions indeed affect the results and complicated the feature location in product variants. The future plan for this study is to examine the extent to which the tangling code of interacted features will affect the results of our approach.

We are also interested to use the variability analysis to help raise the level of variability modeling. In another branch study of our group, we have developed an architecture variability management method and a tool [186]. By the high level's architecture variability management, architecture design and customizations become more intuitive. Additionally, the maintenance efforts are reduced. Now, the current results of our sandwich approach to variability analysis have the potential to be clustered or grouped to a higher level, even to the architecture.

Bibliography

1. Agrawal, R., Imielinski, T. and Swami, A.N., "Mining Association Rules between Sets of Items in Large Databases", *in Proceeding of ACM SIGMOD Conference*, pp. 207-216, 1993.
2. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P. and Lucena, C.J.P., "Refactoring product lines", *in Proceeding of International Conference on Generative Programming and Component Engineering*, pp. 201-210, 2006.
3. Antoniol, G. and Guéhéneuc, Y. G., "Feature identification: an epidemiological metaphor", *IEEE Transactions on Software Engineerin*, vol. 32, no. 9, pp. 627-641, 2006.
4. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D. and Merlo, E., "Recovering traceability links between code and documentation", *IEEE Transactions on Software Engineering*, vol.28, no.10, pp.970-983, October 2002.
5. Apel, S., Kästner, C. and Lengauer, C. "FEATURE-HOUSE: Language-independent, automated software composition", *in Proceedings of International Conference on Software Enginnerring*, pp. 221-231, 2009.
6. Apiwattanapong, T., Orso, A. and Harrold, M.J., "JDiff: A differencing technique and tool for object-oriented programs". *Automated Software Engineering,* vol. 14, no. 1, pp. 3-36, 2007.
7. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D. and Patel-Schneider, P.F., *The Description Logic Handbook: Theory, Implementation, Applications*. Cambridge University Press, Cambridge, UK, 2003.
8. Baker. B., "On finding duplication and near-duplication in large software systems", *In Proceedings of Working Conference on Reverse Engineering*, pp. 86-95, Toronto, Ontario, Canada, July 1995.
9. Balazinska, M., Merlo, E., Dagenais, M., Lague, B. and Kontogiannis, K., "Measuring clone based reengineering opportunities", *in Proceedings of Software Metrics Symposium*, pp. 292-303, USA, 1999.
10. Balazinska. M., Merlo, E., Dagenais, M., Lague, B. and Kontogiannis, K., "Advanced clone-analysis to support object-oriented system refactoring", *in Proceedings of Working Conferencing on Reverse Engineering*, pp. 98-107, 2000.
11. Basit, A.H. and Jarzabek, S., "Detecting higher-level similarity patterns in programs", *In Proceedings of European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 156-165, 2005
12. Basit, H.A. and Jarzabek, S., "A data mining approach for detecting higher-level clones", *IEEE Transactions on Software Engineerin*, vol. 35, no. 4, pp. 497-514, 2009.
13. Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice.* Boston, MA: Addison-Wesley, 1998
14. Bass, L., Clements, P., Cohen, S., Northrop, L., and Withey, J., "Product line practice workshop report", *Technical Report, CMU/SEI-97-TR-003*, 1997.
15. Bassett, P., *Framing Software Reuse - Lessons from Real World*, Prentice Hall, 1997.
16. Batory, D., "Feature-oriented programming and the AHEAD tool suite", *in Proceeding of International Conference on Software Engineering*, pp. 702-703, 2004.
17. Batory, D., "Feature models, grammars, and propositional formulas", *in Proceedings of Software Product Line Conference*, pp. 7-20, 2005.

18. Batory, D., Sarvela, J. N. and Rauschmayer, A., "Scaling step-wise refinement", *IEEE Transactions on Software Engineering*, vol. 30, no. 6, 2004.
19. Batory, D., Sarvela, J.N. and Rauschmayer, A. "Scaling step-wise refinement", *in Proceedings of International Conference on Software Enginnerring*, pp. 187-197, USA, 2003.
20. Baxter, I.D., Yahin, A., Marcelo,L., Sant'Anna,M. and Bier, L., "Clone detection using abstract syntax trees", *in Proceedings of International Conference on Software Maintenance*, pp. 368-377, 1998.
21. Bellon, S., Koschke, R., Antoniol,G., Krinke,J. and Merlo, E., "Comparison and evaluation of clone detection Tools", *IEEE Transactions on Software Engineering*, vol.33, no.9, pp. 577-591, 2007.
22. Benavides, D., Martin-Arroyo, P.T. and Cortes, A.R, "Automated reasoning on feature models", *in Proceedings of International Conference on Advanced Information Systems Engineering*, pp. 491-503, 2005.
23. Berry, M. W., and Browne, M., "Understanding search engines: mathematical modeling and text retrieval", *Society for Industrial and Applied Mathematics*, 2005.
24. Biggerstaff, T., "A perspective of generative reuse", *Annals of Software Engineering*, vol. 5, no.1, pp. 169-226, 1998.
25. Bruntink, M., Deursen, A., Engelen, R. and Tourwé, T., "On the use of clone detection for identifying crosscutting concern code". *IEEE Transactions on Software Engineering*, vol.31, no.10, pp. 804-818, 2005.
26. Buhr, R. J. A. and Casselman, R. S., *Use Case Maps for Object-Oriented Systems*. Prentice Hall, 1996.
27. Buneman, P. and Jajodia S., *Mining Association Rules between Sets of Items in Large Databases*, Washington, D.C., 1993.
28. Chen, K. and Rajlich, V., "Case study of feature location using dependence graph", *in Proceedings of International Conference on Program Comprehension*, pp. 241-249, 2000.
29. Chen, L. and Babar, M.A., "Variability management in software product lines: an investigation of contemporary industrial challenges", *in Proceeding of Software Product Line Conference*, pp. 166-180, 2010.
30. Chen, L., Babar, M.A. and Ali, N., "Variability management in software product lines: a systematic review", *in Proceedings of Software Product Line Conference*, pp. 81-90, 2009.
31. Chi, Y., Muntz, R., Nijssen, S. and Kok, J.N., "Frequent Subtree Mining - An Overview", *Fundamenta Informaticae*, vol. 66, no. 1-2, pp.161-198, January 2005.
32. Clements, P. and Muthig, D. (Editors) Proc. *Workshop on Variability Management at 10th Software Product Line Conf*. Fraunhofer IESE-Report No 152.06/E Version 1.0, Germany, October 15, 2006
33. Clements, P. and Northrop, L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.
34. Cullum, J. K. and Willoughby, R. A., *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, vol. 1: Theory. Chapter 5: “Real rectangular matrices,” Brikhauser, Boston, MA, 1985.
35. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K., "Efficiently computing static single assignment form and the control dependence graph", *ACM Transactions on Programming Languages and Systems*, vol.13, no. 4, pp. 451-490, 1991.
36. Czarnecki, K. and Eisenecker, U., *Generative Programming Methods, Tools, and Applications.* Addison-Wesley, Boston, MA, 2000.
37. Czarnecki, K., Helsen, S. and Eisenecker, U., "Formalizing cardinality-based feature models and their specialization", *Software Process Improvement and Practice*, vol. 10, no. 1, 2005.

38. Dardenne, A., Lamsweerde, A. and Fickas, S., "Goal-directed requirements acquisition", *Science of Computer Programming*, vol. 20, pp.1-2, April 1993.
39. Díaz, O., Trujillo, S. and Anfurrutia F.I., "Supporting production strategies as refinements of the production process", *in Proceedings of Software Product Line Conference*, pp: 210-221, 2005.
40. Deelstra, S., Sinnema, M. and Bosch, J., "Experiences in software product families: problems and issues during product derivation", *in Proceedings of Software Product Line Conference*, pp. 165-182, 2004.
41. Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by latent semantic analysis", *Journal of the American Society for Information Science and Technology*, vol. 41, pp. 391–407. 1990.
42. Demeyer, S., Ducasse, S. and Nierstrasz, O., "Finding refactorings via change metrics", *in Proceeeding of International Conference on Object Oriented Programming, Systems, Languages and Applications*, pp. 166-177, 2000.
43. Dhungana, D., Neumayer, T., Grünbacher, P. and Rabiser, R., "Supporting evolution in model-based product line engineering", *in Proceeeding of Software Product Line Conference*, pp. 319-328, 2008.
44. Dit, B., Revelle, M., Gethers, M. and Poshyvanyk, D., "Feature location in source code: A taxonomy and survey", *Journal of Software Maintenance: Research and Practice*, 2011.
45. Duley, A., Spandikow, C. and Kim, M., "A program differencing algorithm for verilog HDL", *in Proceeding of International Conference on Automated Software Engineering*, pp. 477-486, 2010.
46. Dumais, S. T, "LSI meets TREC: A status report", *in Proceeding of The First Text REtrieval Conference*, D. Harman, Ed. Pp. 137-152. 1992.
47. Eaddy, M., Aho, A. V., Antoniol, G., and Guéhéneuc, Y. G., "CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis", *in Proceedings of International Conference on Program Comprehension*, pp. 53-62, Netherlands, 2008.
48. Eisenbarth, T., Koschke, R., and Simon, D., "Locating features in source code", *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 210 – 224, 2003.
49. Eisenberg, A. D. and De Volder, K., "Dynamic feature traces: finding features in unfamiliar code", *in Proceedings of International Conference on Software Maintenance*, pp. 337-346, 2005.
50. Fellbaum, C., *WordNet: An Electronic Lexical Database*. Cambridge, MA: MIT Press, 1998.
51. Ferber, S., Haag, J. and Savolainen, J., "Feature interaction and dependencies: modeling features for reengineering a legacy product line", *in Proceedings of Software Product Line Conference*, pp. 235-256, 2002.
52. Ferrante, J., Ottenstein, K.J. and Warren, J.D., "The program dependence graph and its use in optimization", *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, 1987.
53. Fluri. B., Wuersch M., Plnzger M. and Gall H., "Change distilling: Tree differencing for fine-grained source code change extraction", *IEEE Transactions on Software Engineerin*, vol. 33, no. 11, pp. 725-743, 2007.
54. Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
55. Gabel, M., Jiang, L. and Su, Z., "Scalable detection of semantic clones", *in Proceedings of International Conference on Software Enginnerring*, pp. 321-330, 2008.

56. Gale, D. and Shapley, L.S., "College admissions and the stability of marriage", *American Mathematical*, vol. 69, pp. 9-14, 1962.

57. Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns*. Addison-Wesley Professional, 1995.

58. Gionis, A.; Indyk, P. and Motwani, R. "Similarity search in high dimensions via hashing", *in Proceedings of Very Large Database Conference*, pp. 518 – 529, 1999.

59. Godfrey, M. and Tu, Q., "Evolution in open source software: a case study", *in Proceedings of International Conference on Software Maintenance*, pp.131-140, 2000.

60. Godfrey, M. and Zou, L., "Using origin analysis to detect merging and splitting of source code entities", *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166-181, 2005.

61. Grahne, G., and Zhu, J., "Efficiently using prefix-trees in mining frequent itemsets", *in Proceeding of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

62. Grant, S., Cordy, J. R., and Skillicorn, D. B., "Automated concept location using independent component analysis", *in Proceedings of Working Conferencing on Reverse Engineering*, pp. 138 - 142, Belgium, 2008.

63. Griss, M.L., Favaro, J. and d'Alessandro M., "Integrating feature modeling with the RSEB", *in Proceedings of International Conference on Software Reuse*, pp. 76-85, 1998.

64. Habermann, A. N., Flon, L., and Cooprider, L., "Modularization and hierarchy in a family of operating systems", *Communications of the ACM*, vol. 19, no. 5, pp. 66–272, 1976.

65. Harman, D, *Ranking algorithms In Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, pp. 363–392. 1992.

66. Hassine, J., Rilling, J., Hewitt, J. and Dssouli, R., "Change impact analysis for requirement evolution using use case maps", *in Proceeding of International Workshop on Principles of Software Evolution*, pp. 81-90, 2005.

67. Higo, Y. and Kusumoto, S., "Enhancing quality of code clone detection with program dependency graph", *in Proceedings of Working Conferencing on Reverse Engineering*, pp. 315-316, 2009.

68. Hill, E., Pollock, L., and Vijay-Shanker, K., "Exploring the neighborhood with Dora to expedite software maintenance", *in Proceedings of International Conference on Automated Software Engineering*, pp. 14-23, 2007.

69. Horwitz, S., "Identifying the semantic and textual differences between two versions of a program", *in Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 234-245, 1990.

70. Jacobson, I., Griss, M. and Jonsson. P., *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Professional, 1997.

71. Jarzabek, S., Bassett, P., Zhang, H. and Zhang, W., "Xvcl: Xml-based variant configuration language", *in Proceeding of International Conference on Software Engineering*, pp: 810–811, USA, 2003.

72. Jarzabek, S., *Effective Software Maintenance and Evolution: Reuse-based Approach*, CRC Press Taylor & Francis, 2007.

73. Jarzabek, S., Ong, W.C. and Zhang, H., "Handling variant requirements in domain modeling". *Journal of Systems and Software*, vol. 68, no. 3, 2003.

74. Jarzabek, S., Xue, Y., "Are clones harmful for maintenance? ", *in Proceeding of International Worshop on Software Cloning*, pp. 73-74, 2010.

75. Jarzabek, S., Xue, Y., Zhang, H. and Lee, Y., "Avoiding some common preprocessing pitfalls with feature queries", *in Proceeding of Asia-Pacific Software Engineering Conference*, pp. 283-290, 2009.

76. Jarzabek, S., Zhang, H., Lee, Y., Xue, Y. and Shaikh, N., "Increasing usability of preprocessing for feature management in product lines with queries", *in Proceeding of International Conference on Software Engineering Companion*, pp. 215-218, 2009.

77. Jiang, L., Su, Z. and Chiu, E., "Context-based detection of clone-related bugs", *In Proceedings of European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 55-64, 2007.

78. Jirapanthong, W. and Zisman, A. "XTraQue: Traceability for product line systems", *Software and System Modeling*, vol. 8, issue 1, pp: 117-144, 2009.

79. Jürgens, E., Deissenboeck, F., Hummel, B. and Wagner, S., "Do code clones matter? ", *in Proceedings of International Conference on Software Engineering*, pp. 485-495, 2009.

80. Kamiya, T., Kusumoto, S. and Inoue, K., "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654-670, 2002.

81. Kang, K., Cohen, S., Hess, J., Nowak, W. and Peterson, S., "Feature-Oriented Domain Analysis (FODA) feasibility study", *Technical Report*. Software Engineering Institute, Carnegie Mellon University, 1990.

82. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E. and Huh, M., "FORM: A feature-oriented reuse method with domain-specific reference architectures". *Annals of Software Engineering*, vol. 5, pp.143-168, 2004.

83. Kang, K.C., Lee, J. and Donohoe, P., "Feature-oriented project line engineering". *IEEE Software*., vol. 19, no. 4, pp. 58-65, 2002.

84. Kapser, C. and Godfrey, M.W., ""Cloning considered harmful" considered harmful", *in Proceedings of Working Conferencing on Reverse Engineering*, pp. 19-28, 2006.

85. Kapser, C. and Godfrey, M.W., "Aiding comprehension of cloning through categorization", *in Proceeding of International Workshop on Principles of Software Evolution*, pp. 85-94, 2004.

86. Karhinen, A., Ran, A. and Tallgren, T., "Configuring designs for reuse", *in Proceedings of International Conference on Software Enginnerring*, pp. 701-710, 1997.

87. Karlsson, J., Olsson, S. and Ryan, K., "Improved practical support for large-scale requirements prioritizing", *Requirements Engineering Journal*, vol. 2, no. 1, pp.51-66, 1997.

88. Kästner, C., and Apel, S. (2008a). "Integrating compositional and annotative ap-proaches for product line engineering", *in Proceeding of GPCE Workshop on Modulariza-tion, Composition and Generative Techniques for Product Line Engineering*, pp. 35–40, 2008.

89. Kästner, C., Apel, S. and Batory, D., "A case study implementing features using AspectJ", *in Proceedings of Software Product Line Conference*, pp.223-232, Japan, 2007.

90. Kästner, C., Apel, S. and Kuhlemann, M., "Granularity in software product lines", *in Proceedings of International Conference on Software Enginnerring*, pp. 311-320, 2008.

91. Kästner, C., Apel, S., and Kuhlemann, M. (2009a). "A model of refactoring physically and virtually separated features", *in Proceeding of International Conference on Generative Programming and Component Engineering*, pp. 157–166, 2009.
92. Kästner, C., Apel, S., Trujillo, S., Kuhlemann, M., and Batory, D. (2009b). "Guaranteeing syntactic correctness for all product line variants: A language-independent approach", *in Proceeding of International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE),* vol. 33 of Lecture Notes in Business Information Processing, pp.175–194. Berlin/Heidelberg: Springer-Verlag, 2009.
93. Kästner, C., Apel, S., ur Rahman, S. S., Rosenmüller, M., Batory, D., and Saake, G. (2009c). "On the impact of the optional feature problem: Analysis and case studies". *in Proceedings of Software Product Line Conference*, pp. 181–190, 2009.
94. Kästner, C., *Aspect-Oriented Refactoring of Berkeley DB*. Master's thesis (Diplomarbeit), University of Magdeburg, 2007.
95. Kellens, A., Mens, K. and Tonella, P., "A survey of automated code-level aspect mining techniques", *Transactions on Aspect Oriented Software Development*, vol. 4, pp. 145-164, 2007.
96. Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M. and Irwin, J., "Aspect-Oriented Programming", *in Proceeding of European Conference on Object-Oriented Programming*, pp. 220-242, Finland, 1997.
97. Kim, M. and Notkin, D., "Discovering and representing systematic code changes", *in Proceedings of International Conference on Software Enginnerring*, pp. 309-319, 2009.
98. Kim, M., Sazawal,V., Notkin, D. and Murphy, G.C.. "An empirical study of code clone genealogies", *In Proceedings of European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 187-196, 2005.
99. Komondoor, R., and Horwitz, S., "Using slicing to identify duplication in source code", *in Proceedings of International Symposium on Static Analysis*, pp. 40-56, 2001.
100. Koschke, R. and Quante, J., "On dynamic feature location", *in Proceeding of International Conference on Automated Software Engineering*, pp. 86-95, 2005.
101. Koschke, R., "Survey of research on software clones", *Duplication, Redundancy, and Similarity in Software*, 2006.
102. Kothari, J., Denton, T., Mancoridis, S., and Shokoufandeh, A., "Reducing program comprehension effort in evolving software by recognizing feature implementation convergence", *in Proceedings of International Conference on Program Comprehension*, pp. 17-26 , Canada, 2007.
103. Krinke, J., "A study of consistent and inconsistent changes to code clones", *in Proceedings of Working Conferencing on Reverse Engineering*, pp. 170-178, 2007.
104. Krinke, J., "Identifying similar code with program dependence graphs", *in Proceedings of Working Conferencing on Reverse Engineering*, pp. 301-310, 2001.
105. Krueger, C.W., "Practical strategies and techniques for adopting software product lines", *in Proceedings of International Conference on Software Resue*, pp. 349-350, 2002.
106. Laddad, R., *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.

107. Lague, B., Proulx, D., Mayrand, J., Merlo, E.M. and Hudepohl, J., "Assessing the benefits of incorporating function clone detection in a development process", *In Proceedings of the International Conference on Software Maintenance*, pp. 314–321. IEEE Computer Society, 1997.

108. Lanubile, F. and Mallardo, T., "Finding function clones in web applications", *In Proceedings of European Conference on Software Maintenance and Reengineering*, pp. 379-386, Benevento, Italy, March 2003.

109. Li,Z., Lu,S., Myagmar, S. and Zhou, Y., "CP-Miner: Finding copy-paste and related bugs in large-scale software code", *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176-192, 2006.

110. Liebig, J., Apel, S., Lengauer, C., Kästner, C. and Schulze, M., "An analysis of the variability in forty preprocessor-based software product lines", *in Proceeding of International Conference on Software Engineering*, pp. 105-114, 2010.

111. Linden, F., Schmid, K. and Rommes, E., *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

112. Liu, C., Chen, C., Han, J. and Yu, P.S., "GPLAG: Detection of software plagiarism by program dependence graph analysis", *in Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 872-881, USA, 2006.

113. Liu, D., Marcus, A., Poshyvanyk, D. and Rajlich, V., "Feature location via information retrieval based filtering of a single scenario execution trace", *in Proceedings of International Conference on Automated Software Engineering*, pp: 234-243, 2007.

114. Loh, A. and Kim, M., "LSdiff: a program differencing tool to identify systematic structural differences", *in Proceedings of International Conference on Software Engineering Companion*, pp. 263-266, 2010.

115. Lormans, M., "Monitoring requirements evolution using views", *in Proceeeding of European Conference on Software Maintenance and Reengineering*, pp. 349–352, 2007.

116. Lucia, A.D., Fasano, F., Oliveto, R. and Tortora, G., "Recovering traceability links in software artifact management systems using information retrieval methods", *ACM Transaction on Software Engineering and Methodology*, vol. 16, no. 4, 2007.

117. Maoz, S., Ringert, J.O. and Rumpe, B., "ADDiff: semantic differencing for activity diagrams", *in Proceeding of ACM SIGSOFT Foundation of Software Engineering*, pp. 179-189, 2011.

118. Marcus A. and Maletic, J.I., "Recovering documentation-to-source-code traceability links using Latent Semantic Indexing", *in Proceeding of International Conference on Software Engineering*, pp.125-137, 2003.

119. Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An information retrieval approach to concept location in source code", *in Proceedings of Working Conferencing on Reverse Engineering*, pp.214-223, Netherlands, 2004.

120. Mayrand, J., Leblanc, C. and Merlo, E., "Experiment on the automatic detection of function clones in a software system using metrics", *in Proceedings of International Conference on Software Maintenance*, pp. 244, 1996.

121. Mendonca, M., Branco, M. and Cowan, D., "S.P.L.O.T. - software product lines *online tools", in Proceeding of ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, pp. 761-762, 2009.

122. Mendonca, M., Wasowski, A., Czarnecki, K. and Cowan, D., "Efficient compilation techniques for large scale feature models", *in Proceeding of International Conference on Generative Programming and Component Engineering*, pp. 13-22, ACM, 2008.

123. Mens, T., "A state-of-the-art survey on software merging", *IEEE Transactions on Software Engineerin*, vol. 28, no. 5, pp. 449-462, 2002.

124. Myers, E.W., "An O(ND) difference algorithm an its variations", *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986.

125. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M. and Zave, P., "Matching and merging of statecharts specifications", *in Proceedings of International Conference on Software Engineering*, pp. 54 – 64, 2007.

126. Ommering, R.C., "Building product populations with sofware components", *in Proceeding of International Conference on Software Engineering*, pp. 255-265, 2002.

127. Parnas, D. L., "On the design and development of program families", *IEEE Transactions on Software Engineering*, vol. 2, no. 1, pp. 1-9, 1976.

128. Pearse, T. T., and Oman, P. W., "Experiences developing and maintainingsoftware in a multi-platform environment", *in Proceedings of International Conference on Software Maintenance*, pp. 270–277. 1997.

129. Pettersson, U. and Jarzabek, S., "Industrial experience with building a web portal product line using a lightweight, reactive approach", *in Proceeding of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 326-335, 2005.

130. Peng, X., Xing, Z., Tan, X., Yu, Y. and Zhao, W., "Iterative context-aware feature location", *in Proceeding of International Conference on Software Engineering*, pp. 900-903, 2011.

131. Person, S.J., *Differential Symbolic Execution*, Doctor's thesis, University of Lincoln, Nebraska, 2009.

132. Pigoski, Thomas M., *Practical Software Maintenance*. New York: John Wiley & Sons, 1996.

133. Pohl, K., Böckle, G. and van der Linden, F.J., *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition, September 2005.

134. Poshyvanyk, D. and Marcus, D., "Combining formal concept analysis with information retrieval for concept location in source code", *in Proceedings of International Conference on Program Comprehension,* pp. 37-48, Canada, 2007.

135. Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval", *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420-432, June 2007.

136. Poshyvanyk, D., Marcus, A., Rajlich, V., Guéhéneuc, Y.-G. and Antoniol, G., "Combining probabilistic ranking and latent semantic indexing for feature identification*", in Proceeding of International Conference on Program Comprehension*, pp. 137-148, 2006.

137. Prehofer, C., "Feature-Oriented Programming: A fresh look at objects", *in Proceeding of European Conference on Object-Oriented Programming*, pp. 419-443, 1997.

138. Rajapakse, D.C. and Jarzabek, S., "Towards generic representation of web applications: solutions and trade-offs", *Software Practice and Experience*, vol. 39, no. 5, pp. 501-530, 2009.

139. Rajlich, V. and Wilde, N., "The role of concepts in program comprehension", *in Proceeding of International Workshop on Program Comprehension*, pp. 271-278, 2002.

140. Ramalingam, G. and Reps, T., "Semantics of program representation graphs", *Technical Report*, University of Wisconsin Madison, 1989.

141. Refstrup, J. G., "Adapting to change: Architecture, processes and tools: A closer look at HP's experience in evolving the Owen software product line", *in Proceeding of Software Product Line Conference*, Keynote presentation, 2009

142. Revelle, M. and Poshyvanyk, D., "An exploratory study on assessing feature location techniques", *in Proceedings of International Conference on Program Comprehension*, pp. 218-222, 2009.

143. Rosenmüller, M., Siegmund, N., Schirmeier, H., Sincero, J., Apel, S., Leich, T., Spinczyk, O. and Saake, G., "FAME-DBMS: Tailor-made data management solutions for embedded systems", *in Proceeding of Software Engineering for Tailor-made Data Management*, pp. 1-6, 2008.

144. Roy, C.K. and Cordy, J.R., "A survey on software clone detection research", *Technical Report* 2007-541, Queen's University, 2007.

145. Roy, C.K. and Cordy, J.R., "Scenario-based comparison of clone detection techniques", *in Proceedings of International Conference on Program Comprehension*, pp. 153-162, 2008.

146. Roy, C.K., "Detection and analysis of near-miss software clones", *in Proceedings of International Conference on Software Maintenance*, pp. 447-450, 2009.

147. Segura, S., Benavides, D., Ruiz-Cortés, A., and Trinidad, P., "Automated merging of feature models using graph transformations", *Generative and Transformational Techniques in Software Engineering II*, pp. 489 - 505, 2007.

148. Shepherd, D., Fry, Z., Gibson, E., Pollock, L., and Vijay-Shanker, K., "Using natural language program analysis to locate and understand action-oriented concerns", in *in Proceedings of Conference on Aspect Oriented Software Development*, pp. 212-224, 2007.

149. Silverstein, C., Henzinger, M.R., Marais, H. and Moricz, M., "Analysis of a very large web search engine query log". *SIGIR Forum*, vol. 33, no.1, pp. 6-12, 1999.

150. Simmons, S., Edwards, D., Wilde, N., Homan, J., and Groble, M., "Industrial tools for the feature location problem: an exploratory study", *Journal of Software Maintenance: Research and Practice*, vol. 18, no. 6, pp. 457-474, 2006.

151. Sincero, J. and Schröder-Preikschat, W., "The Linux kernel configurator as a feature modeling tool". *in Proceeding of Software Product Line Conference*, pp. 257-260, 2008.

152. Sinnema, M. and Deelstra, S., "Classifying variability modeling techniques", *Information & Software Technology*, vol. 49, no. 7, pp. 717-739, 2007.

153. Sinnema, M., Deelstra, S., Nijhuis, J. and Bosch, J., "COVAMOF: A framework for modeling variability in software product families", *in Proceedings of Software Product Line Conference*, pp. 197–213, 2004.

154. Smaragdakis, Y. and Batory, D., "Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs ", *ACM Transaction on Software Engineering and Methodology*, vol. 11, no. 2, pp. 215-255, 2002.

155. Spencer, H. and Collyer, G., "#ifdef considered harmful, or portability experience with C news," *USENIX, Summer Technical Conference*, pp. 185-197, San Antonio, Texas, June, 1992.

156. Steimann, F., Pawlitzki, T., Apel, S., and Kästner, C., "Types and modularity for implicit invocation with implicit announcement", *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 1, 2010.

157. Sun, J., Liu, Y. and Dong, J.S., "Model Checking CSP Revisited: Introducing a Process Analysis Toolkit", *in Proceeding of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*,, pp. 307-322, 2008.

158. Sun, J., Liu, Y., Dong, J.S. and Pang, J., "PAT: Towards Flexible Verification under Fairness", *in Proceeding of International Conference on Computer Aided Verification*, pp. 709-714, 2009.

159. Sun, J., Zhang, H., Li, Y. and Wang, H., "Formal Semantics and Verification for Feature Modeling", *in Proceeding of International Conference on Engineering of Complex Computer Systems*, pp. 303-312, 2005.

160. Svahnberg, M., Gurp, J. and Bosch, J., "A taxonomy of variability realization techniques", *Software: Practice and Experience*, vol. 35, no. 8, 2005.

161. Tarr, P., Ossher, H., Harrison, W. and Sutton Jr. S.M., "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *in Proceeding of International Conference on Software Engineering*, pp. 107-119, 1999.

162. Thüm, T., Batory, D., and Kästner, C., "Reasoning about edits to feature models", *in Proceedings of International Conference on Software Enginnerring*, pp. 254-264, 2009.

163. Treude, C., Berlik, S., Wenzel, S. and Kelter, U, "Difference computation of large models", *in Proceeding of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 295-304, 2007.

164. Trifu, M, "Using dataflow information for concern identification in object-oriented software systems", *In Proceedings of European Conference on Software Maintenance and Reengineering*, pp. 193-202, 2008.

165. Ueda, Y., Kamiya, S., Kusumoto, S. and Inoue, K., "Gemini: Maintenance support environment based on code clone analysis", *in Proceedings of IEEE International Software Metrics Symposium*, pp. 67-76, 2002.

166. Ueda, Y., Kamiya, T., Kusumoto, S. and Inoue. K., "On detection of gapped code clones using gap locations", *In Proceedings of Asia-Pacific Software Engineering Conference*, pp. 327-336, Australia, 2002.

167. Wang, H., Li, Y., Sun, J., Zhang, H. and Pan, J., "Verifying Feature Models using OWL", *Journal of Web Semantics*, vol. 5, no. 2, Elsevier, pp. 117-129, 2007.

168. Wang, S., Lo, D., Xing, Z. and Jiang, L., "Concern localization using information retrieval: An empirical study on Linux kernel", *in Proceedings of Working Conferencing on Reverse Engineering*, pp. 92-96, 2011.

169. Wilde, N. and Scully, M., "Software reconnaissance: mapping program features to code", *Software Maintenance: Research and Practice*, vol. 7, pp. 49-62, 1995.

170. Wong, E., Gokhale, S, Horgan, J. and Trivedi, K., "Locating program features using execution slices". *In Proceedings of IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, pp.194–203, 1999.

171. Xing, Z. and Stroulia, E., "Differencing logical UML models", *Journal of Automated Software Engineering*, vol. 14, no. 2, pp. 215-259, 2007.

172. Xing, Z. and Stroulia, E., "Refactoring detection based on UMLDiff change-facts queries", *in Proceedings of Working Conferencing on Reverse Engineering*, pp. 263-274, 2006.

173. Xing, Z., Xue, Y. and Jarzabek, S, "CloneDifferentiator: Analyzing clones by differentiation", *in Proceeding of International Conference on Automated Software Engineering*, pp. 576-579, 2011.

174. Xing, Z., Xue, Y. and Jarzabek, S., "Distilling useful clones by contextual differencing ", *Technical Report*, National University of Singapore, 2012.

175. Xing, Z.: GenericDiff, "A general framework for model comparison", *in Proceeding of International Conference on Automated Software Engineering*, pp. 135-138, 2010.

176. Xue, Y., Jarzabek, S., Ye, P., Peng, X. and Zhao, W., "Scalability of variability management: An example of industrial practice and some improvements", *in Proceeding of International Conference on Software Engineering and Knowledge Engineering*, pp. 705-710, 2011.

177. Xue, Y., "Reengineering legacy software products into software product line based on automatic variability analysis", *in Proceeding of International Conference on Software Engineering*, pp. 1114-1117, 2011.

178. Xue, Y., Xing, Z. and Jarzabek, S., "CloneDiff: Semantic differencing of clones*", in Proceeeding of International Workshop on Software Cloneing*, pp. 83-84, 2011.

179. Xue, Y., Xing, Z. and Jarzabek, S., "Feature location in a collection of product variants", *in Proceedings of Working Conferencing on Reverse Engineering,* to appear, 2012.

180. Xue, Y., Xing, Z. and Jarzabek, S., "Understanding feature evolution in a family of product variants", *in Proceedings of Working Conference on Reverse Engineering*, pp. 109-118, 2010.

181. Yang, W., "Identifying syntactic differences between two programs", *Software Practice and Experience*, vol. 21, no. 7, pp. 739-755, 1991.

182. Ye, P., Peng, X, Xue, Y. and Jarzabek, S, "A case study of variation mechanism in an industrial product line", *in Proceeding of International Conference on Software Reuse*, pp. 126-136, USA, 2009.

183. Yoshida, N., Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K., "On refactoring support based on code clone dependency relation", *in Proceedings of International Software Metrics Symposium*, pp. 16, 2005.

184. Zhang, Y., Basit, H.A., Jarzabek, S., Anh, D. and Low, M., "Query-based filtering and graphical view generation for clone analysis", *in Proceedings of International Conference on Software Maintenance*, pp. 376-385, 2008.

185. Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., "SNIAFL: Towards a static non-interactive approach to feature location", *ACM Transaction on Software Engineering and Methodology*, vol. 15, no. 2, pp. 195-226, 2006.

186. Zhu, J., Peng, X., Jarzabek, S., Xing, Z., Xue, Y. and Zhao, W., "Improving product line architecture design and customization by raising the level of variability modeling", *in Proceedings of International Conference on Software Reuse*, pp. 151-166, 2011.

187. Zowghi, D. and Offen, R., "A logical framework for modeling and reasoning about the evolution of requirements", *in Proceeding of International Requirements Engineering Conference*, pp. 247–259, 1997.

188. Ant : http://ant.apache.org/

189. Berkeley DB Java Edition, 2012. http://www.oracle.com/database/berkeley-db/je/index.html

190. BigLever. GEARS. http://www.biglever.com/, 2009.

191. CIDE: http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/, 2012
192. Concept Explorer: http://sourceforge.net/projects/conexp/, 2011
193. Eclipse JDT project, http://www.eclipse.org/jdt/, 2010
194. FEATUREHOUSE: http://www.infosun.fim.uni-passau.de/spl/apel/FH/
195. Linux Kernel, http://www.kernel.org/
196. Java Annotation: http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html
197. Java Conditional Compilation: http://c2.com/cgi/wiki?ConditionalCompilationInJava
198. Parameterized Test Patterns for Microsoft Pex, http://research.microsoft.com/en-us/projects/pex/patterns.pdf, 2011.
199. Pure systems. pure::variants. http://www.puresystems.com, 2009
200. Semantic Vectors: http://code.google.com/p/semanticvectors/, 2011
201. SMV model checker. http://www.cs.cmu.edu/~modelcheck/smv.html.
202. Sparse: https://sparse.wiki.kernel.org/index.php/Main_Page#Sparse_-_a_Semantic_Parser_for_C, 2011
203. Sun Java IO, http://java.sun.com/javase/6/docs/api/java/io/, 2010
204. WALA:http://wala.sourceforge.net/wiki/index.php/Main_Page, 2010
205. XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, http://xvcl.comp.nus.edu.sg