Software Features Extraction from Object-Oriented Source Code Using an Overlapping Clustering Approach

Imad Eddine Araar
Department of Mathematics and Computer Science
Larbi Ben M'hidi University, Oum el Bouaghi, Algeria
E-mail: imad.araar@gmail.com

Hassina Seridi
Electronic Document Management Laboratory (LabGED)
Badji Mokhtar-Annaba University, P.O. Box 12, 23000 Annaba, Algeria
E-mail: seridi@labged.net

Keywords: feature model, software product line, overlapping clustering, reverse engineering, program analysis

Received: January 16, 2016

For many decades, numerous organizations have launched software reuse initiatives to improve their productivity. Software product lines (SPL) addressed this problem by organizing software development around a set of features that are shared by a set of products. In order to exploit existing software products for building a new SPL, features composing each of the used products must be specified in the first place. In this paper we analyze the effectiveness of overlapping clustering based technique to mine functional features from object-oriented (OO) source code of existing systems. The evaluation of the proposed approach using two different Java open-source applications, i.e. "Mobile media" and "Drawing Shapes", has revealed encouraging results.

Povzetek: Prispevek vpelje novo metodo generiranja spremenljivk za ponovno uporabo objektno usmerjenih sistemov.

1 Introduction

A software product line, also known as software family, is "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [6]. A feature represents a prominent or distinct aspect that is visible to the user, a quality, or a system characteristic [16]. There are two types of features: (1) the commonalities, that must be included in all products, and (2) the variabilites, which are shared by only some of them. A feature model (FM) provides a detailed description of the commonalities and variabilities, specifying all the valid feature configurations. Driven by the software industrial requirements, i.e. cost and time-to-market, several organizations have therefore chosen to convert to a SPL solution. Such a migration can be achieved using one of three major adoption strategies: proactive, reactive, extractive [19]. Using a proactive approach, the organization analyzes, designs and implements a SPL to support all anticipated products (which are within the scope of the SPL). With the reactive approach, organizations develop their SPLs in an incremental manner. This strategy is appropriate when the requirements of new products in the SPL are somehow unpredictable. The extractive approach is used to capitalize on existing software systems by extracting their commonalities and variabilities. Since the proactive strategy is the most expensive and exposed to risks [19], most researchers are now interested in reengineering commonalities and variabilities from existing systems.

On this matter, the type of artifacts to be used in the SPL reengineering process seems of great importance, since it strongly impacts the quality of the results, as well as the level of user involvement in this process. Most of the existing SPL extractive approaches use various types of artifacts such as FMs of existing systems, requirements documents or other additional data. However, most of the requirement documents are written in natural language and, therefore, suffer from several problems such as scalability, heterogeneity and ambiguity [21, 26]. The software documentation on the other hand may be obsolete after making several changes to the code without updating its documentation. In addition, it happens that some systems are not yet equipped with a FM, knowing that it is indispensable throughout the entire SPL development cycle. In order to obtain a FM of an old system, commonalities and variabilities that characterize such a system must be identified and documented. The manual construction of a FM is an expensive and greedy task [33]. Hence, assisting this process would be of great help.

In this article, we propose a new approach which mines features from Java source code of an existing system for disentangling stakeholder goals. Compared to the existing approaches that mine features, the novelty of

the proposed approach is that it provides a generic and reusable features catalogue for a software variant instead of generating a single FM which is specific to a set of product variants. We use an overlapping clustering algorithm in order to minimize the information loss. In the proposed approach, Java programs' elements constitute the initial search space. By conducting a static analysis on the target system, we define a similarity measure that enables to proceed through a clustering process. The result is subgroups of elements each of which represents a feature implementation. The number of clusters is automatically calculated during the mining process which decreases the expert involvement.

The remainder of this paper is structured as follows: Section 2 presents the state of the art and motivates our work. Section 3 describes, step by step, our proposed approach for features reengineering using Java source code of existing systems. Section 4 reports the experimentation and discusses threats to the validity of the proposed approach. Finally, Section 5 concludes and provides perspectives for this work.

2 State of the art

This section carries out a survey of leading papers describing the work carried out so far which are related to reverse engineering feature models. Depending on the type of artifacts used as input, one can distinguish two main subgroups of studies that aim to extract FMs: (1) documentation-based approaches and (2) source code-based approaches.

2.1 Documentation-based techniques

Extraction of FMs from legacy systems can benefit from the existing experience in reverse engineering works. The approach proposed by Graaf et al. [14] consists in migrating an existing architecture to a SPL using automatic model transformations. The transformation rules used are defined using ATL (Atlas Transformation Language). Thus, migration can only be possible if the variability is defined by a meta-model.

Niu et al. [22] present a new approach based on clustering, information theory, and natural language processing (NLP) to extract features by analyzing functional requirements. They use an overlapping clustering algorithm. NLP technique is used to define the similarity between the attributes of FRP (functional requirement profiles); FRPs are abstractions of functional requirements. However, the FRPs and their attributes must be prepared manually which implies a considerable human effort.

Rashid et al. [26] propose a technique based on NLP and clustering for automatic construction of FM from heterogeneous requirements documents. However, the FM generated by their approach was of great size compared with a manually created FM. Hence, the intervention of an expert is always needed to perform pre and/or post-processing. The authors explain that this problem is caused by the used clustering algorithm and irrelevant information contained in the inputs.

Haslinger et al. [15] present an algorithm that reverse engineers a FM for a given SPL from feature sets which describe the characteristics each product variant provides. The features used were obtained by decomposing FMs retrieved from an online repository. Experiments have shown that the basic FMs calculated by this algorithm are identical to the initial models retrieved from the repository.

Ziadi et al. [33] propose an automatic approach to identify features for a set of product variants. They assume that all product variants use the same vocabulary to name the program elements. However, given that their approach uses UML class diagrams as inputs, it doesn't consider the method body. In addition, their approach identifies all common features as a single mandatory feature (a maximal set), that is shared by all the product variants.

Ryssel et al. [27] present a technique based on formal concept analysis (FCA) that analyzes incidence matrices containing matching relations as input and creates FMs as output. The matrix describes parts of a set of function-block-oriented models. Compared to other FCA-based approaches, their approach uses optimization techniques to generate the lattices and FMs in a reasonable time.

2.2 Source code-based techniques

There exist very few studies that have addressed the problem of reverse engineering FMs using the source code as a starting point. Kästner et al. [17] propose a tool, Colored IDE (currently known as the CIDE tool), to identify and mark code fragments that correspond to features. However, the process is still manual and it depends on the experience of the tool user. The CIDE tool seems to be more useful in feature-oriented refactoring tasks.

Loesch et al. [20] propose a new approach for restructuring an existing SPL. Their FCA-based approach consists in analyzing real products configuration files used in a given SPL, and building a lattice that provides a classification of variable features usage in product variants. This classification can be used as a formal basis in interactive tools in order to restructure the variabilities.

Paskevicius et al. [23] propose a framework for an automated generation of FM from Java source code using static analysis and conceptual clustering. The approach uses as input the dependency graph (DG) of a targeted software system. The DG is transformed into a distance matrix that must be analyzed using the CobWeb algorithm [12] in order to create a features hierarchy. This latter is used to generate the final FM as Feature Description Language (FDL) descriptors and as Prolog rules. Their approach may be useful during the partial configuration of a given system, for example, to derive a light version of a system.

Al-Msie'deen et al. [25] propose an approach for generating FMs from Java source code of a set of existing systems. They suppose that the analyzed systems use the same vocabulary to name the program elements, i.e. product variants belonging to the same SPL. First,

they explore candidate systems to extract OO building elements (OBE). These items are then analyzed combining FCA analysis, Latent Semantic Indexing (LSI), and structural similarity to identify features. Their approach has given good results. However, the analyzed systems may not use the same vocabulary to name OBEs, which means that the used lexical similarity cannot be always reliable.

2.3 Synthesis

As we can see from the preceding sections, most of the existing approaches that address the problem of reverse engineering FMs from existing systems are semi-automatic, and use the documentation and textual descriptions as inputs. However, such a practice involves several challenges. Although the input data type in a given approach is strongly linked to its purpose and usage type, the abstraction level and formalization of such data must also be considered. Ziadi et al. [33] use as input the parts of UML class diagram, which is likely to omit many details related to the variability in the program implementation. FM generated from requirements in [26] was imprecise and very large. The authors justified their finding by the heterogeneous and especially textual nature of the inputs. These latter are likely to be filled with imprecise language commonly used in conversations. Besides the ambiguity, scalability is a problem in the context of SPL. Indeed, we can find a significant number of documents associated with a given product variant, each of which is very large in terms of size. Other works such that given by Haslinger et al. [15] use a set of FM of existing systems in order to derive the SPL' feature model. However, existing systems do not always have a FM, and even if exists it may not be up to date and, therefore, does not truly reflect the variabilities of these systems.

Regarding the used technique, the majority of approaches use classification, since it is the most suitable for the problem of FM generation. Some approaches have used a clustering technique to generate FMs. Paskevicius et al. [23] consider the hierarchy generated by the Cobweb algorithm as a FM, while there is simply not enough information in the input data in order to decide one preferred hierarchy. Moreover, the use of a simple clustering algorithm to generate disjoint groups of program elements can cause information loss, since a program element can be part of more than one feature (crosscutting concerns). Niu et al. [22], address the overlapping problem using an overlapping partitioning algorithm called OPC [5]. Nevertheless, the OPC algorithm requires four parameters to be specified by the user, which significantly minimizes the automation of the task.

Besides clustering, many researchers have used FCA analysis to extract FMs while taking into account the overlap problem. However, there is a limit in the use of FCA. Indeed, not only FCA does not assure that the generated features (formal concepts) are disjoint and cover the entire set of entities [30], but it is also exposed to the information loss problem. For example, in [25], cosine similarity matrices are transformed into a (binary) formal context using a fixed threshold. The information loss caused by such a sharp threshold usage may affect the quality of the result, as claimed by the authors in [2]. The REVPLINE approach proposed by Al-Msie'Deen et al. [2, 25] generates SPL features using as inputs the source code of product variants. They suppose that analyzed products are developed with copy-paste technique, i.e. they use the same vocabulary. However, if this assumption does not hold, it is therefore essential to have a separate FM for each product variant in order to generate the SPL feature model.

3 A tool support for automatic extraction of features

This section presents the main concepts, hypotheses and techniques used in the proposed approach for mining features from source code.

3.1 Goal and core assumptions

The overall aim of the proposed method is to identify all feature implementations for a given software product, based on static analysis of source code. In fact, We recognize that it is essential to have, for every software, a FM which is up to date and reflects the changes that were made to the source code over time. The generated features can be used for documenting a given system as well as for reverse engineering a FM for a SPL. We adhere to the classification given by [16] which distinguishes three categories of features: functional, operational and presentation features. In this article we focus on the identification of functional features; functional features express how users can interact with a software system.

The functional features are implemented using OO program elements (PEs), such as packages, classes, class attributes, methods or elements of method bodies. We also consider that the PEs can be classified in two categories: (1) *atomic program elements* (APE), and (2) *composite program elements* (CPE). An APE is a basic construction element in the program (a variable or a method). A CPE is a composition of atomic and/or composite PEs (i.e. a class or a package). A dependency is a relation between two PEs. An element A depends on B if A references B. For example a method $A()$ uses a variable B or calls a method $B()$. Given a dependency graph $G = (V, E)$, a cluster is defined as a sub-graph $\grave{G} = (\grave{V}, \grave{E})$ whose nodes are tightly connected, i.e. cohesive. Such clusters are considered as functional feature implementations. We suppose also that feature implementations may overlap; a given PE may be shared by the implementations of several features simultaneously.

In addition, since a class represents the main building unit in OO languages, we assume that a generated feature is represented by at least one class. Indeed, a class is generally referred to as a set of responsibilities that simulates a concept or a feature in the application domain [9]. This hypothesis has been

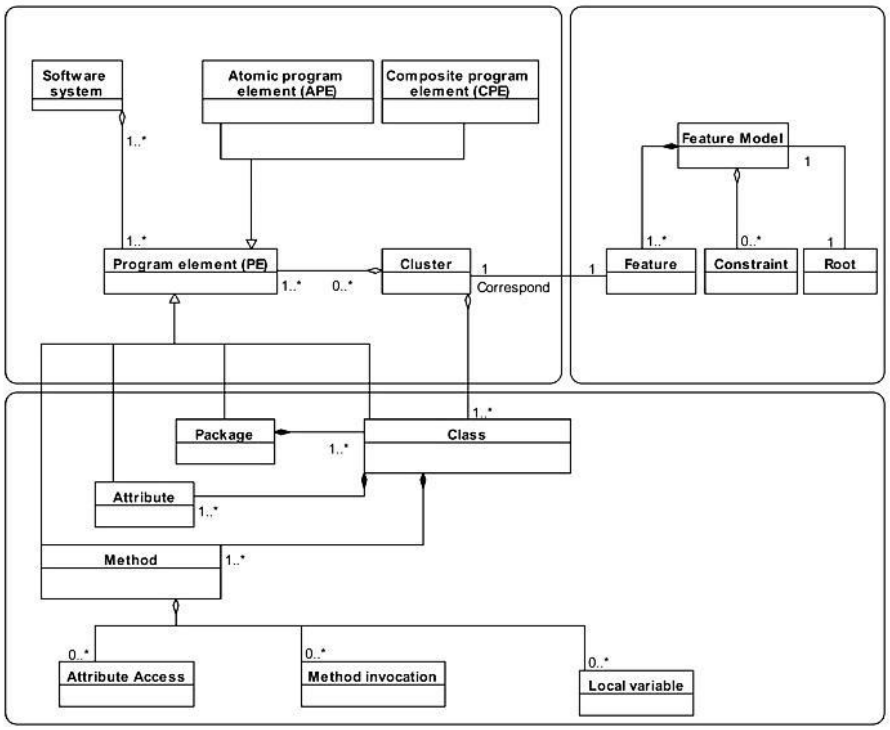

Figure 1: A meta-model to map source code to features.

checked by experiments carried out in [25]. For the sake of simplicity, we make no distinction between classes per se and abstract classes. Taking into account this assumption about classes, interfaces must be pruned from the PEs set. In fact, as a method body must anyway be redefined at the class level when implementing an interface, then considering interfaces in the inputs will not provide additional information to the overall mining process. Consequently, pruning interfaces from the initial PEs set will not affect the validity of our assumptions.

Furthermore, since a software system's features are associated with its behavior, we decided to keep only those PEs which are created by developer to implement the system's specific features. For example, a linked list is a concept from the solution domain which may be implemented in the source code, yet it is not a specific feature of the system. All the concepts we defined for mining features are illustrated in the "program to feature mapping model" of Figure 1.

3.2 Feature mining step by step

This section presents, in a detailed way, the feature reengineering process. Input data were prepared using the method described by Paskevicius et al. [23] while introducing necessary modifications to comply with assumptions and techniques used in our proposed approach. The architecture of our proposed approach for mining features from source code is given in Figure 2.

3.2.1 Extraction of program elements and dependencies

Dependencies of the candidate system was modelled using an oriented dependency graph $G = (F, D)$, such that F is a set of vertices which represent PEs, and D is a set of dependencies. The dependency graph G was generated and saved into an XML file by analyzing ".class" files using *DependencyExtractor*. This latter is a part of a toolbox called *JDependencyFinder*[1]. The use of the Java byte code instead of the source code facilitates the analysis of existing systems whose source code is not available. Moreover, sometimes source code lacks information like, for example, how the compiled code will be organized in execution containers (Jar files). Such information is usually defined in the scripts executed during compilation [7]. The choice of using a dependency graph as input is justified by the nature of the problem as well as the granularity of the processed entities. In fact, we try to build clusters of PEs, i.e. feature implementations, based on functional dependencies between these PEs; a feature implementation is characterized by a strong functional dependency (intra-cluster cohesion) between its composing PEs.

DependencyExtractor was executed through the command line by combining three of its parameters: [-*class-filter*], [-*minimize*] and [-*filter-excludes*]. The [-*minimize*] parameter was used to remove redundant dependencies. In fact, it is often the case that an explicit dependency in the code can be implied from another explicit dependency in that code. Such dependencies do not add anything to the overall connectivity of the graph and, therefore, must be removed. The second parameter, i.e. [-*class-filter*], allows us to select only those dependencies going to/from classes. These latter represent, as explained earlier, the main construction

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

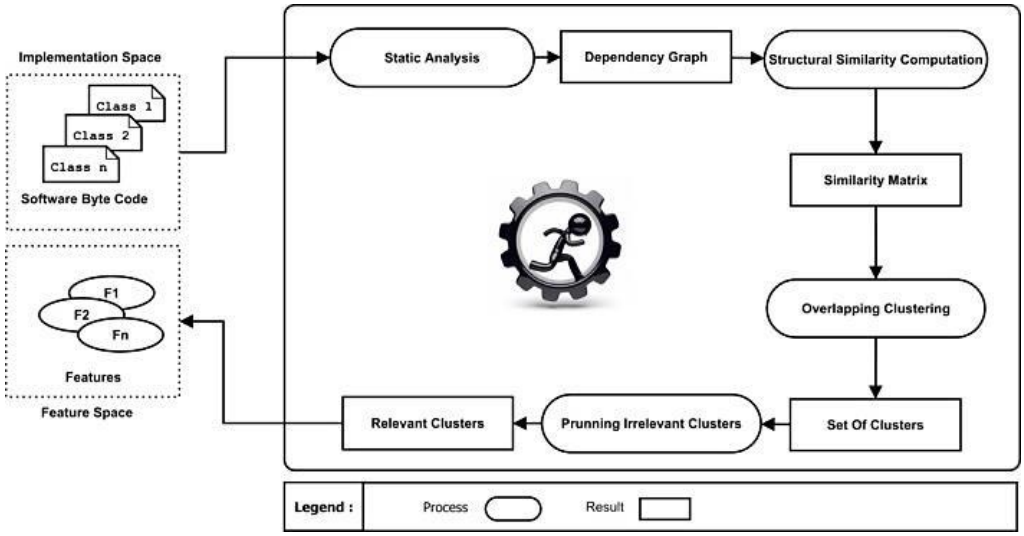

Figure 2: The feature mining process.

units of feature implementations. The choice of using such type of dependencies instead of only considering inter-class dependencies is justified by the fact that such kind of mixed dependencies are likely to enrich the training set. Finally, the third parameter [*-filter-excludes*] is used to remove graph vertices (resp. dependencies) which represents language specific libraries, such that "Java.*" and "javax.*".

3.2.2 Constructing the similarity matrix

The dependency graph generated in the previous step is used in this phase to build a similarity matrix. A native structural distance-based measure was used to evaluate similarity between each pair of vertices. Firstly, the dependency graph G was expressed, as an adjacency matrix C of the size $|F|$, such that $c_{ij} = 1$ means that there exists an explicit dependency from i to j. In order to describe indirect dependencies, the matrix C has been converted into a distance matrix M of size $|F|$ using Floyd-Warshall's all pairs shortest path algorithm (after Floyd [13] and Warshall [31]), such that m_{ij} is equal to the shortest path distance between program elements i and j. The two matrices C and M are asymmetric because the dependency graph G is directed. Given that the used clustering algorithm operates on an undirected graph, the asymmetric distance table M was converted into a symmetric table $\grave{M}$, such that $\grave{m}_{ij} = \grave{m}_{ji} = Min\ (m_{ij}, m_{ji})$. In addition, it is difficult to estimate the similarity between two PEs using absolute distance $\grave{m}_{ij}$. Hence, absolutes distances matrix was converted into a normalized similarity matrix in such a way that two given PEs have a similarity $S(i,j) = 0$ if there is no path between them, and a similarity $S(i,j) = 1$ if they are identical.

3.2.3 Building feature implementations

After preparing the training set using program dependencies, OclustR algorithm [24] was then executed on that data to generate a set of clusters of PEs. Each calculated cluster is considered as the implementation of a single feature. The OclustR passes through two main stages: (1) initialization step, and (2) the improvement step.

The main idea of the initialization phase is to produce a first set X of sub-graphs, i.e. *ws-graphs*, that covers the graph; in this context, each ws-graph consists in a candidate cluster. Afterward, during the improvement phase, a post-processing is performed on the initial clusters in order to reduce their number and overlap. To do this, the set X is analyzed to remove ws-graphs which are considered as *less useful*. These latter are pruned by merging the sets of their vertices with those of a chosen ws-graph.

Formally, let $O = \{PE_1, PE_2, \ldots, PE_n\}$ be a set of PEs. The OclustR algorithm uses as input an undirected and weighted graph $\tilde{G}_\beta = (V, \tilde{E}_\beta, S)$, such that $V = O$, and there is an edge $(v,u) \in \tilde{E}_\beta$ iff $v \neq u$ and $S(v,u) \geq \beta$, with $S(PE_1, PE_2)$ is a symmetric similarity function and $\beta \in [0,1]$ is a user-defined threshold; Each edge $(v,u) \in \tilde{E}_\beta$ is labeled with the value of $S(v,u)$. We assume that each PE must be assigned at least to one cluster, even if the similarity between that element and the cluster's center is very small. Thus, there is an edge $(v,u) \in \tilde{E}_\beta$ iff $v \neq u$ and $S(v,u) > 0$. Consequently, we are sure that every PE in the training set will be assigned to at least one cluster. The OclustR algorithm doesn't need, henceforth, any input parameter, which increase the task automation. The user still can select other values for the parameter β in order to generate features with more fine-grained granularity, so he can have multiple views of the analyzed system with different abstraction levels.

3.2.4 Pruning irrelevant clusters

When constructing the dependency graph, we have selected only those dependencies whose composing nodes contains at least one class, which means that the resulting clusters will be composed of APEs and/or CPEs. Taking into account that classes are considered as the main construction units of feature implementations,

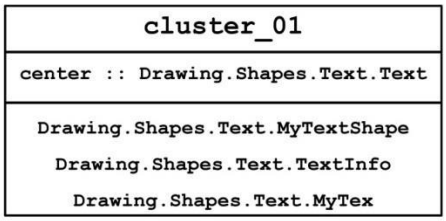

Figure 3: Example of a mined feature.

clusters which contain only APEs are, consequently, considered as irrelevant and pruned from the result set. In the case of a mixed content cluster, each APE has been replaced by its source class. Indeed, APEs involved in such clusters are only used to provide an optional detailed view. Hence, the final result is a set of relevant clusters composed only by classes. Figure 3 represents a feature (i.e. cluster) mined by our proposed approach for the Drawing Shapes case study (see Section 4.1.1). Given that a cluster computed using OclustR is mainly a ws-graph, it's, consequently, defined by his name which is a unique reference given by the system, his center, and his satellites.

DependencyExtractor tool. Experiments were made on Windows 7 based PC with Intel i5-4200U processor and 8G of RAM.

4.1.1 Case studies

In order to validate the proposed approach, we conducted experiments on two different Java open-source applications: *Mobile media*[3] and *Drawing Shapes*[4]. The advantage of having two case studies is that they implement variability at different levels. In addition, the corresponding documentations and FMs are available which facilitate the comparison with our results. Moreover, using these two case studies we target two different categories of FMs: (1) a flat FM which is related with the Drawing Shapes case study, and (2) a nested FM related with the Mobile Media case study. Figure 4 and Figure 5 present the corresponding FMs, following the notation proposed by Ferber et al. [10].

The Drawing Shapes SPL represents a small case study (version 5 consists of 8 packages and 25 classes with about 0,6 KlOC). The Drawing Shapes application allows a user to draw seven different kinds of shapes in a

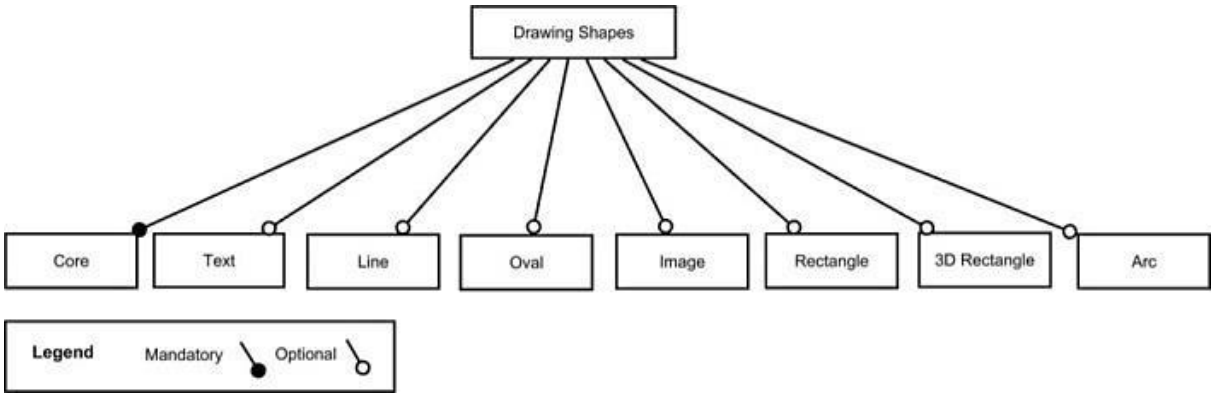

Figure 4: The Drawing Shapes FM.

4 Experimentations

In this section, the experimental setup is described and subsequently, the empirical results are presented in detail, together with a discussion of possible limitations and threats to validity of this study.

4.1 Experimental setup

We fully implemented the steps described in Section 3.2 as a Java tool. We tried to develop the features inference engine as an independent component so that it can be used in a generic and efficient manner. The feature inference engine reads dependency graphs generated using a static analyzer, and seeks to discover software features. In our experiments, we tested our implemented tool on two Java programs, but software written in other OO languages (i.e. C++ or C#) can also be analyzed using our tool, if their dependency graphs are delivered in XML files respecting the DTD[2] used by the variety of colors. The user chooses the shape and the color, and then presses the mouse button and drag the mouse to create the shape. The user can draw as many shapes as desired. The Drawing Shapes software variants were developed based on the copy paste modify technique. In this example, we use version 5 (the full version) which supports *draw 3D rectangle*, *draw rectangle*, *draw oval*, *draw string* and *draw arc* features, together with the *core* one.

The Mobile Media [32] SPL is a benchmark used by researchers in the area of program analysis and SPL research [2, 11, 29]. It manipulates photo, music, and video on mobile devices, such as mobile phones. Mobile Media endured seven evolution scenarios, which led to eight releases, comprising different types of changes involving mandatory, optional, and alternative features, as well as non-functional concerns. In this example, we used the sixth release (R6) which contains OO implementation of all the optional and mandatory features for managing Photos. The used release of

[2] http://depfind.sourceforge.net/dtd/dependencies.dtd

[3] http://homepages.dcc.ufmg.br/~figueiredo/spl/icse08/

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

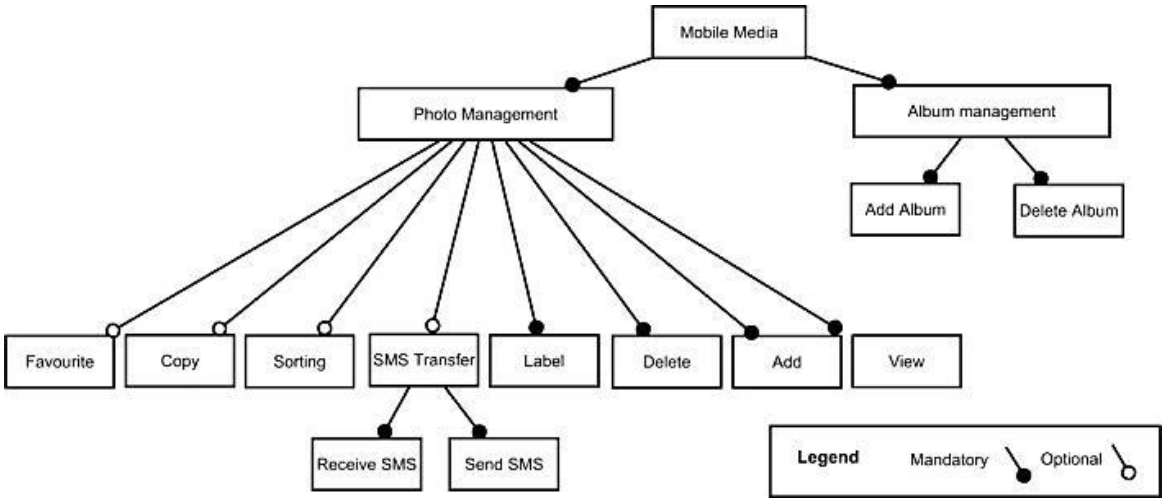

Figure 5: The Mobile Media FM.

Mobile Media consists of 9 packages, 38 classes with about 3 KLOC.

The PEs used to implement the *Exception handling* feature in this Mobile Media release were discarded from the entry set in the pre-processing phase. Indeed, as mentioned before, this paper deals only with functional feature implementations. Additionally, we used the default similarity threshold value for OclustR, i.e. $\beta = 0$.

4.1.2 Evaluation measures

The accuracy and performance of the proposed method is evaluated using an external validation measure. The F-measure [8] is a well-known tool in the information retrieval (IR) domain, which can also be used as a measure for flat clustering quality. We assume that for a given set of program elements $O = \{o_1, o_2, \dots, o_n\}$ we have both: the real, true partition of this set $L = \{L_1, L_2, \dots, L_{K^L}\}$ (we can call L sets as classes, K^L is the number of classes) and clustering partition, the result of OclustR algorithm $C = \{C_1, C_2, \dots, C_{K^C}\}$ (C sets as clusters, K^C is the number of clusters), in order to compute how similar they are.

For the two aforementioned case studies, the true partitions (i.e. listing of real features' PEs) were prepared manually by inspecting the documentation provided by their authors.

F-measure is a mixture of two indices: precision (P), which measures the homogeneity of clusters with respect to a priori known classes, and recall (R), that evaluates the completeness of clusters relatively to classes. A higher precision shows that almost all the cluster elements correspond to expected class. A lower recall, in the other hand, indicates that there are various actual elements that were not retrieved. The convenience of F-measure at this stage of our work is justified by the behavior of this metric. Indeed, F-measure computes the quality of every cluster independently with respect to each class, which allows us to automatically determine the most representative cluster for each class in the gold standard.

Having the previously introduced notation, *precision* of cluster C_i with regard to class L_j is computed as follows:

$$P(C_i, L_j) = \frac{|C_i \cap L_j|}{|C_i|}$$

Recall of cluster C_i with respect to class L_j is computed as follows:

$$R(C_i, L_j) = \frac{|C_i \cap L_j|}{|L_j|}$$

Thus, the F value of the cluster C_i with respect to class L_j is the combination of these two:

$$F(C_i, L_j) = \frac{2 \times P(C_i, L_j) \times R(C_i, L_j)}{P(C_i, L_j) + R(C_i, L_j)}$$

Hence, the F-measure for a cluster C_i is the highest of F values obtained by comparing this cluster with each of known classes:

$$F(C_i) = \max_{L_j \in L} F(C_i, L_j)$$

Despite the fact that the usage of F-measure at this point decreases the expert involvement, this metric seems to be inappropriate when assessing the overall effectiveness of an overlapping clustering solution. According to [3], F-measure does not always detect small improvements in the clustering distribution, and that might have negative implications in the system evaluation/refinement cycles. The authors in [3] have proposed a new metric, i.e. $FBCubed$, that gives a good estimation of the clustering system effectiveness while taking into account the overlapping among clusters. Thus, we decided to evaluate the overall accuracy and performance of our proposed method using $FBCubed$. The $FBCubed$ is calculated using the BCubed Precision and BCubed Recall metrics as proposed in [3]. The BCubed Precision and BCubed Recall are based on the Multiplicity Precision and Multiplicity Recall metrics respectively; which are defined as:

$$MP(o_1, o_2) = \frac{Min(|C(o_1) \cap C(o_2)|, |L(o_1) \cap L(o_2)|)}{|C(o_1) \cap C(o_2)|}$$

$$MR(o_1, o_2) = \frac{Min(|C(o_1) \cap C(o_2)|, |L(o_1) \cap L(o_2)|)}{|L(o_1) \cap L(o_2)|}$$

Where o_1 and o_2 are two program elements, $L(o_1)$ are the classes associated to o_1, $C(o_1)$ are the clusters associated to o_1. $MP(o_1, o_2)$ is the Multiplicity Precision of o_1 wrt o_2, such that o_1 and o_2 share at least one cluster. $MR(o_1, o_2)$ is the Multiplicity Recall of o_1 wrt o_2, such that o_1 and o_2 share at least one class.

Let $D(o_i)$ be the set of PEs that share at least one cluster with o_i including o_i. The BCubed Precision metric of o_i is defined as:

$$BCubed_{Precision}(o_i) = \frac{\sum_{o_j \in D(o_i)} MP(o_i, o_j)}{|D(o_i)|}$$

Let $H(o_i)$ be the set of PEs that share at least one class with o_i including o_i. The BCubed Recall metric of o_i is defined as:

$$BCubed_{Recall}(o_i) = \frac{\sum_{o_j \in H(o_i)} MR(o_i, o_j)}{|H(o_i)|}$$

The overall BCubed Precision of the clustering solution, denoted as $BCubed_{Precision}$, is computed as the average of the BCubed precision of all PEs in the distribution O; the overall BCubed recall of the clustering solution, denoted as $BCubed_{Recall}$, is defined analogously but using the BCubed recall of all PEs. Finally, the FBCubed measure of the clustering solution is computed as the harmonic mean of $BCubed_{Precision}$ and $BCubed_{Recall}$ as follows:

$$FBCubed = \frac{2 \times BCubed_{Precision} \times BCubed_{Recall}}{BCubed_{Precision} + BCubed_{Recall}}$$

4.2 Results and discussions

Our implemented tool has derived features quickly from the used case studies in a reasonable amount of time (about 1 second). Table 1 summarizes the obtained results. For the sake of readability, we manually associated feature names to clusters, based on their content. Of course, this does not impact the quality of our results.

Firstly, we observe that the F values obtained in the Mobile media case study have been greatly influenced by the recall values. The precision values however, remain high for the majority of the features, indicating the relevance of their composing PEs.

These observations can be explained by the operating mechanism of OclustR, which produces partitions having strong cohesion and low overlap. Indeed, we assumed that the features are simulated using one or more classes at the code level. Considering this assumption, implementations of Mobile Media features are therefore strongly overlapped. Indeed, many classes are shared by the implementations of numerous features at the same time, because of the use of several design patterns by the Mobile Media authors such as *Model-Vue-Controller (MVC)* and *Chain of responsibility*. For example, the classes *PhotoController*, *ImageAccessor* and *ImageData* encapsulate all the photo management methods. Thus, features such as *send picture*, *sorting* and *add picture* share most of their PEs (i.e. classes) and, therefore, have reached a low Recall and F values. The *Splash screen* feature however, has not suffered from this problem since all functionalities related to this feature were encapsulated in distinguishable classes. Thus, the *splash screen* feature obtained a maximum F value.

Feature	Evaluation metrics		
	P %	R %	F %
Drawing Shapes (version 5)			
Core	100	57	73
Drawing Text	100	100	100
Drawing Oval	100	100	100
Drawing Line	100	100	100
Drawing Rectangle	100	100	100
Drawing Image	100	100	100
Drawing Arc	100	100	100
Drawing 3D-Rectangle	11	33	17
Mobile Media (version 6)			
Splash screen	100	100	100
SMS transfer	64	41	50
Photo management	100	35	52
Album management	40	22	29
View photo	54	58	56
Edit photo Label	70	39	50
Add photo	100	9	17
Delete photo	40	40	40
Favourites	100	8	15
Sorting	100	8	15
Add album	67	22	33
Delete album	38	38	38
Send photo	100	14	25
Receive photo	40	17	24

Table 1: Features mined from Mobile Media and Drawing Shapes softwares.

The conclusions drawn from the Mobile Media analysis comply with those obtained for the Drawing Shapes case study. The Drawing Shapes features are slightly overlapped, so we reached a maximum *F* value, except for the *3D-Rectangle* and *Core* features. The low F value of *3D-Rectangle* is caused by the low cohesion between its classes in the source code level. In fact, the manual verification of the *3D-Rectangle* source code revealed that the Drawing Shapes author has intentionally or accidently caused this low cohesion by creating *3D-Rectangle* classes without creating dependencies between them (i.e. classes instantiations are missing). Thus, the PEs involved in the implementation of *3D-Rectangle* were scattered throughout the results, so that a low *F* value was reached. However, in the case of *Core* feature, some relevant PEs were mined and mistakenly mapped to another cluster so

that the *F* value was slightly affected. Anyway, the F-measure for this latter is 0.73 on average which is an acceptable value.

The evaluation of overlapping clustering solution for each of the case studies using the FBCubed metric clearly confirms the above findings (see Table 2). The overall BCubed Precision of the Mobile Media case study remains high and the low value of the Bcubed Recall affected consequently the global FBCubed value. In the other hand, balanced and high values were reached for the Drawing Shapes' Bcubed metrics.

Software	BCubed Precision	Bcubed Recall	FBCubed
Drawing Shapes	80%	67%	73%
Mobile Media	78%	25%	38%

Table 2: The BCubed evaluation results of the overlapping clustering solutions.

Despite F-measure and FBCubed proved to be appropriate, our implemented tool generates a small number of clusters most of which are relevant. This latter characteristic makes our results easily understandable and effectively handleable by the user. Table 3 shows that our tool generated 50% of relevant clusters for Drawing Shapes case study, and 88% for Mobile Media.

Software	Mined clusters	Relevant clusters	Relevance ratio
Drawing Shapes	16	8	50%
Mobile Media	16	14	88%

Table 3: Number of reliable mapping.

The results show that our proposed approach has generated a reasonable number of features with an acceptable precision. According to the case studies that we have conducted, our proposed method operates efficiently when dealing with programs having flat FMs. Unlike documentation-based approaches [15, 26, 33], our proposed approach relies on source code as it seems to be the most reliable source that can capitalize on the knowledge and the expertise of experts who participated in the development of the analyzed systems. Since they consist of sets of program elements, the features generated by the proposed approach are of a more formal nature which, thereby, facilitates their interpretation and further manipulation.

The proposed approach provided a feature catalogue instead of generating one preferred hierarchy. This latter characteristic, together with the formal nature of the generated features, represents the key strength of our proposed approach, so it can operate in a generic and efficient manner. Thus, results obtained by our proposed method can be manipulated by other complementary tools in order to get additional information and, therefore, construct a reliable FM. Our proposed method is also complementary to other approaches such as software transplantation [4]. Inspired from human organ transplantation, this latter works by isolating the code of a useful feature in a "donor" program and transplanting this "organ" to the right "vein" in software lacking the feature. Our proposed approach can act in such case by delimiting and extracting a feature before its transplantation.

In addition, our proposed method addressed the information loss problem that characterizes most of FCA-based methods by the usage of a similarity measure. This latter can be tuned by the user to change the granularity of outputs. Compared to other clustering approaches [22, 23], our proposed approach used a new partitioning algorithm that provides overlapping clusters in an efficient manner. The user involvement was negligible during all the steps of our experimentation. Hence, the method can be potentially very useful and it can save stakeholder from a lot of effort and time required to specify features composing each software variant during the SPL reverse engineering task.

4.3 Threats to validity

There is a limit to the use of Floyd's algorithm to infer similarity between PEs. In fact, the complexity determined by this algorithm is of $O(n^3)$ [18]. In addition, precomputing all the shortest paths and storing them explicitly in a huge dependency matrix seems to be challenging in terms of space complexity. These two factors affect the applicability of the proposed approach on larger software systems. In fact, even if computing shortest paths is a well-studied problem, exact algorithms cannot be adopted for a massive dependency graph.

Moreover, as illustrated above, the OclustR algorithm manages clusters' overlapping but still represents several restrictions when dealing with clusters that are strongly overlapped, which limits the usability of the proposed approach to systems with a nested FM. Another problem related to OclustR that may affect the results accuracy happens when a given class, i.e. an abstract class, is inherited by most of the system classes and, thus, will be considered as the center c of a ws-graph ($G_c^\star$) having all the inheriting classes as satellites. Hence, during the improvement phase, each ws-graph having as center one of the $G_c^\star$ satellites will be judged as irrelevant. In this case, we call $G_c^\star$ a *predatory ws-graph* and his center c a *predatory center*. Such predatory cluster phenomena may affect the results accuracy.

Finally, structural distance-based measure used in the proposed approach still has some restrictions. Indeed, we used a simple technique to compute similarity between PEs based on the number of steps on the shortest path relating them in the graph. Even that such a strategy has given acceptable results, it still has some limitations since it does not consider the multiplicity in paths (i.e. connectivity) between a pair of nodes.

5 Conclusion and perspectives

In this paper, we proposed a new method for reverse engineering software functional features from source code. We used dependencies that exist between program

elements at the source code level in order to apply a graph clustering algorithm in an efficient way. We tested our implemented tool to recover features from source code of two existing java programs. We obtained promising results that are consistent with the main objectives of our study, which makes the proposed approach useful for mining features from software source code.

In future work, we would like to improve output quality using other overlapping clustering techniques, in order to overcome the aforementioned OclustR limitations. We also plan to automatically extract mined feature names, based on features contents, in order to facilitate their interpretation and manipulation in further tasks.

Furthermore, in order to tackle the complexity problem when computing structural distance-based measures, we plan to use approximation methods based on random walks [1], such as random walk with restart. Besides complexity optimization, random walk-based measure provides a result that is different from that of the shortest-path measure because the multiplicity in paths between a pair of nodes is also leveraged when computing similarity. Such a measure is likely to enhance accuracy of our results and to reduce the effects of the predatory clusters phenomena.

Moreover, since software features are associated with its behavior, we intend to enrich input data using dynamic information. Indeed, even if they are based on different operating strategies, dynamic and static analyses can be complementary in certain points [28]. Hence, a dynamically collected data is likely to enhance the result set by additional information.

6 References

[1] C. C. Aggarwal (2015), "*Similarity and Distances*", *Data Mining: The text book*, pp. 63-91: Springer International Publishing.

[2] R. Al-Msie'deen *et al.* (2014), "*Automatic Documentation of [Mined] Feature Implementations from Source Code Elements and Use-Case Diagrams with the REVPLINE Approach*", International Journal of Software Engineering and Knowledge Engineering, vol. 24, n°. 10, pp. 1413-1438.

[3] E. Amigó *et al.* (2009), "*A comparison of extrinsic clustering evaluation metrics based on formal constraints*", Information Retrieval, vol. 12, n°. 4, pp. 461-486.

[4] E. T. Barr *et al.* (2015), "*Automated software transplantation*", in International Symposium on Software Testing and Analysis Baltimore, MD, USA, pp. 257-269.

[5] Y.-L. Chen, and H.-L. Hu (2006), "*An overlapping cluster algorithm to provide non-exhaustive clustering*", European Journal of Operational Research, vol. 173, n°. 3, pp. 762-780.

[6] P. Clements, and L. Northrop (2001), "*Software product lines: practices and patterns*", Addison-Wesley.

[7] J. Dietrich *et al.* (2008), "*Cluster analysis of Java dependency graphs*", in Proceedings of the 4th ACM symposium on Software visualization, Ammersee, Germany, pp. 91-94.

[8] K. Draszawka, and J. Szymański (2011), "*External Validation Measures for Nested Clustering of Text Documents*", *Emerging Intelligent Technologies in Industry*, Studies in Computational Intelligence pp. 207-225: Springer Berlin Heidelberg.

[9] H. Eyal-Salman, A.-D. Seriai, and C. Dony (2013), "*Feature-to-Code Traceability in Legacy Software Variants*", in 39th EUROMICRO Conference on Software Engineering and Advanced Applications Santander, Spain, pp. 57-61.

[10] S. Ferber, J. Haag, and J. Savolainen (2002), "*Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line*", in Proceedings of the Second International Conference on Software Product Lines, pp. 235-256.

[11] E. Figueiredo *et al.* (2008), "*Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability*", in 30th International Conference on Software Engineering, Leipzig, Germany, pp. 261-270.

[12] D. Fisher (1987), "*Knowledge Acquisition Via Incremental Conceptual Clustering*", Machine Learning, vol. 2, n°. 2, pp. 139-172.

[13] R. W. Floyd (1962), "*Algorithm 97: Shortest path*", Communications of the ACM, vol. 5, n°. 6, pp. 345.

[14] B. Graaf, S. Weber, and A. van Deursen (2006), "*Migrating supervisory control architectures using model transformations*", The 10th European Conference on Software Maintenance and Reengineering. pp. 153-164.

[15] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed (2011), "*Reverse Engineering Feature Models from Programs' Feature Sets*", in 18th Working Conference on Reverse Engineering, Limerick, Ireland, pp. 308-312.

[16] K. Kang *et al.* (1990), *Feature Oriented Domain Analysis (FODA) Feasibility Study*, Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, United States.

[17] C. Kästner, M. Kuhlemann, and D. Batory (2007), "*Automating feature-oriented refactoring of legacy applications*", in ECOOP Workshop on Refactoring Tools, pp. 62-63.

[18] S. Khuller, and B. Raghavachari (2009), "*Basic graph algorithms*", *Algorithms and Theory of Computation Handbook, Second Edition, Volume 1*, Chapman & Hall/CRC Applied Algorithms and Data Structures series: Chapman & Hall/CRC.

[19] C. W. Krueger (2002), "*Easing the Transition to Software Mass Customization*", *Software Product-Family Engineering : Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, Lecture Notes in Computer Science pp. 282-293: Springer Berlin / Heidelberg.

[20] F. Loesch, and E. Ploedereder (2007), "*Restructuring variability in software product lines*

using concept analysis of product configurations", Proceedings of 11th European Conference on Software Maintenance and Reengineering CSMR '07. pp. 159-168.

[21] B. Meyer (1985), "*On Formalism in Specifications*", IEEE Software, vol. 2, n°. 1, pp. 6-26.

[22] N. Niu, and S. Easterbrook (2008), "*On-Demand Cluster Analysis for Product Line Functional Requirements*", Proceedings of 12th International Software Product Line Conference SPLC '08. pp. 87-96.

[23] P. Paskevicius *et al.* (2012), "*Automatic Extraction of Features and Generation of Feature Models from Java Programs*", Information Technology and Control, vol. 41, n°. 4, pp. 376-384.

[24] A. Pérez-Suárez *et al.* (2013), "*OClustR: A new graph-based algorithm for overlapping clustering*", Neurocomputing, vol. 121, pp. 234-247.

[25] R. Al-Msie'Deen *et al.* (2013), "*Mining Features from the Object-Oriented Source Code of Software Variants by Combining Lexical and Structural Similarity*", in IEEE 14th International Conference on Information Reuse & Integration, Las Vegas, NV, USA, pp. 586-593.

[26] A. Rashid, J. C. Royer, and A. Rummler (2011), "*Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way*", Cambridge University Press.

[27] U. Ryssel, J. Ploennigs, and K. Kabitzsch (2011), "*Extraction of feature models from formal contexts*", in Proceedings of the 15th International Software Product Line Conference, Volume 2, Munich, Germany, pp. 1-8.

[28] E. Stroulia, and T. Systä (2002), "*Dynamic analysis for reverse engineering and program understanding*", ACM SIGAPP Applied Computing Review, vol. 10, n°. 1, pp. 8-17.

[29] L. P. Tizzei *et al.* (2011), "*Components meet aspects: Assessing design stability of a software product line*", Information and Software Technology, vol. 53, n°. 2, pp. 121-136.

[30] P. Tonella, and A. Potrich (2007), "*Reverse Engineering of Object Oriented Code*", Springer-Verlag New York, 1 ed.

[31] S. Warshall (1962), "*A Theorem on Boolean Matrices*", Journal of the ACM (JACM), vol. 9, n°. 1, pp. 11-12.

[32] T. J. Young (2005), "*Using aspectj to build a software product line for mobile devices*", Master Thesis, The University of British Columbia.

[33] T. Ziadi *et al.* (2012), "*Feature Identification from the Source Code of Product Variants*", Proceedings of 16th European Conference on Software Maintenance and Reengineering (CSMR). pp. 417-422.