Feature Model Extraction from Product Source Codes based on the Semantic Aspect

Jihen Maazoun[1], Nadia Bouassida[2], Hanêne Ben-Abdallah[1] and Abdelhak-Djamel Seriai[3]
[1]*Mir@cl Laboratory, Faculté des Sciences Economiques et de Gestion , Sfax University, Sfax, Tunisia*
[2]*Mir@cl Laboratory, Institut Supérieur d'Informatique et de Multimédia, Sfax University, Sfax, Tunisia*
[3]*LIRMM Laboratory, University of Montpellier 2, Montpellier, France*

Keywords: SPL, Semantic Name Correspondence, Feature Diagram, Feature Identification From Source Code.

Abstract: Software Product Lines can be constructed through either a top-down or bottom-up process. A top-down process begins by a domain analysis where variabilities are specified then it derives the product line. It is especially interesting for the creation of new product lines. However, in practice, SPL are often set up after several similar product variants have been in use. This practice prompted the search for bottom-up processes that start from an analysis of existing product variants to identify the product line. The proposed bottom-up processes rely on two hypotheses: the product variants use the *same* vocabulary to name elements in their source code, and the product variants have *very similar/identical* structures. However, while the names represent the application domain of the products, when different developers were involved in the development of the product variants, the naming assumption becomes too restrictive. Furthermore, the variants' code structures are often different when developed separately and even when one variant is derived from another through several modifications. To loosen these two hypotheses, this paper proposes a bottom-up approach that integrates the semantic aspect of the product variants when extracting the SPL feature model. In addition, a second contribution of our approach is its capability to identify *automatically* the constraints among the identified features.

1 INTRODUCTION

A Software Product Line (SPL) (Clements and Northrop, 2001) is a set of systems that share a group of manageable features. A feature is seen as an end-user, visible characteristic of the system (Kang et al., 1990). The features of an SPL can be used in variable combinations to derive product variants in the SPL. To represent the variability in an SPL, a feature model (or diagram) is used to specify the SPL variants and variation points; it indicates the features and the constraints (and, or, require, etc) relating the features to one another. A feature model is constructed by the development processes of SPL either in a bottom-up (*cf.,* (Ziadi et al., 2012), (She et al., 2011)) or top-down (*cf.,* (Ziadi, 2004)) approach.

A top-down development process starts with a domain analysis to construct the feature model of an SPL. Thus, this process is driven by the functional requirements towards the definition of alternative solutions. It is best applied when the application domain has not yet been sufficiently explored. However, this type of processes is time consuming and requires guidelines (so far undefined) for the domain requirements analysis. On the other hand, a bottom-up process starts from the code of a set of products in a given domain and it identifies their common and variable features. The purpose of examining sample products is to create a generic reusable SPL that is understandable and easy to reuse.

In recent years, several researchers examined the extraction of features and/or feature models from product source codes. Existing feature identification approaches (Ziadi et al., 2012), (Al-Msie'Deen et al., 2012), (Salman et al., 2012) (Acher et al., 2013) rely on two essential hypotheses: all source codes use the *same* vocabulary to name packages, classes, attributes and methods in their source codes; and the product variants have *very similar/identical* structures. These assumptions stem from their way of seeing how the product variants were created: essentially through "copy" and "paste" operations which, indeed, preserve the names and cause little structuring changes. However, these approaches cannot be applied in the

Maazoun J., Bouassida N., Ben-Abdallah H. and Seriai A..
Feature Model Extraction from Product Source Codes based on the Semantic Aspect.
DOI: 10.5220/0004486701540161
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-EA-2013), pages 154-161
ISBN: 978-989-8565-68-6
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

general setting where an SPL should be constructed from product variants that were produced by different developers, and/or product variants that endured so many modifications that the "same names and structure" assumptions are violated. For instance, a class in one product can be represented in a second product through two classes where the attributes and methods of the original class are distributed. A second example of product variability is when a class in one product was moved from one package to another package. For these simple examples, existing feature identification approaches would fail. Besides the two restrictive assumptions, existing bottom-up SPL construction approaches have limitations in the identification of the constraints relating identified features: Some approaches offer no guidance in the identification of the constraints (*cf.*, (Ziadi et al., 2012)), while others manage to identify some constraints without a certitude on their types, *e.g.*, they do not distinguish between the AND and Require constraints. However, the constraints are as important as the features in an SPL since they are the means to derive product variants from an SPL.

In this paper, we propose a bottom-up SPL construction approach that both accounts for the differences in the names and structures of the source code products, and identifies automatically the feature constraints along with the features. Our approach was inspired from (Al-Msie'Deen et al., 2012); it fine-tuned Formal Concept Analysis (FCA)(Ganter and Wille, 1996) and Latent Semantic Indexing (LSI) (Binkley and Lawrie, 2011) by using semantic analysis to bypass the naming assumption and alleviate the structure assumption. In addition, it confirms the feature constraint types by using semantic criteria.

The remainder of this paper is organized as follows. Section 2 overviews currently proposed approaches for feature identification from source code of product variants. Section 3 presents our technique for feature extraction using the semantics. In addition, it illustrates it through an example of an SPL for mobile phones. Finally, Section 4 summarizes the paper and outlines our future work.

2 RELATED WORK

As mentioned in the introduction, an SPL is often modeled in terms of a feature model. Before examining existing approaches for SPL construction, we find it necessary to first overview the concept of feature models.

2.1 Feature Models

Feature models are a popular means to express requirements in a domain at an abstract level. They are used to describe variable and common properties of products in a product line, and to derive and validate configurations of software systems. As introduced by the FODA method (Kang et al., 1990) and by (Czarnecki and Eisenecker, 2000), a feature model represents a hierarchy of properties of domain concepts. A feature is a prominent or distinctive quality or characteristic of a software system or systems (Kang et al., 1990).

Feature models have also been used to improve program comprehension. In most reengineering activities, the source code is the only reliable source of information. Feature models allow designers to bridge the gap between the concrete code and the fairly abstract information of documents such architecture design. They are applied in methods for supporting reverse engineering in a hypothesis-verification procedure (Riebisch, 2003).

A Feature model has a tree structure where each node represents a feature. Feature variability is represented by the arcs and groupings of features. There are two different types of feature groups: Mandatory, Optional (see Table 1).

Table 1: Different types of feature groups.

Feature	Definition	Notation
Mandatory	Child feature is obligatory.	
Optional	Child feature is optional.	

In addition to the parental relationships between features, constraints between the nodes across-tree are allowed. The five most common cross-tree constraints are: Xor, Or, Require, And, Exclude (see Table 2)

Note that since the features are identified from the code source, then each feature can be documented by the corresponding source code parts. This documentation is essential in the creation of a new product variant from a feature model. Note also that a feature can be either simple/elementary like a package and a

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 2: Different constraints in feature groups.

Constraint	Definition	Notation
Or	At least one of the sub-features must be selected	
Xor	One of the sub-features must be selected	
Require	The selection of A in a product necessitates the selection of B	A → B
And	A and B must be part of the same product.	A B
Excludes	A and B cannot be part of the same product	A - - - → B

class, or composed of several elements like {package, Class}, {package, Class,attribute, method}...

2.2 Feature Model Extraction Approaches

Several approaches were proposed to extract feature models. Some of them extract feature models from product descriptions. For example, Acher et al., (Acher et al., 2010) propose to reverse engineer feature models from the documentation of products; and (Acher et al., 2013) propose to extract feature models from the product configurations. Other approaches propose to extract feature models from source code but for different purposes (Lozano, 2011). One main purpose of extracting feature models is to obtain information to understand and possibly refactor the SPL. For instance, (Rubin and Chechik, 2012) aims at refactoring existing, closely related products into a product line. Their approach compares the product elements and measures their degree of similarity based on which elements can be merged as features.

A second purpose of feature model extraction is to analyze and understand the evolution of SPL *cf.,* (Xue, 2011), (Loesch and Ploedereder, 2007). For example, (Xue, 2011) aims at assisting analysts in detecting changes to product features during their evolution. They propose a model differencing algorithm to identify evolutionary changes that occurred to features of different product variants. On the other hand, Loesch et al. (Loesch and Ploedereder, 2007) note that obsolete variable features are not removed after an SPL evolution; thus, they treat the problem of restructuring variability in an SPL that was degraded in its evolution. For this purpose, they use Formal Concept Analysis (FCA) to construct a lattice that classifies the usages of variable features in real products.

Furthermore, some works were interested in feature model extraction for reverse engineering and maintenance purposes. As an example, (She et al., 2011) propose an approach that constructs the feature model once the features have been identified. This approach takes as inputs a set of feature names, descriptions and dependencies, and it uses a set of heuristics to find the *hierarchies* among the features. It can identify feature groups (*i.e.* features used together), mandatory features and features that imply or exclude other features. However, it cannot identify alternative groups (*Or-groups*).

Several works investigated feature model extraction from the source code of products in order to construct the SPL ((Ziadi et al., 2012), (Al-Msie'Deen et al., 2012),(Paskevicius et al., 2012)). For instance, (Ziadi et al., 2012) propose an approach that first abstracts the input products in SoCPs (Sets Of Construction Primitives) and, secondly, it identifies features by determining common and intersecting SoCPs. This approach was validated using two case studies: a banking example and the Argo-UML software product line (Couto et al., 2011). The obtained results show that the approach can handle products with variable names of classes, methods and attributes. However, this approach does not examine the body of the methods. In addition, the produced feature model contains only one mandatory feature and optional features; it can identify neither separated mandatory features, nor alternative features and their related constraints such as the mutual exclusion.

On the other hand, Al-Msie'Deeen (Al-Msie'Deen et al., 2012) propose an approach based on the definition of the mapping model between OO elements and feature model elements. This approach uses Formal Concept Analysis (FCA) to cluster similar OO elements into one feature. It uses Latent Semantic Indexing (LSI) to define a similarity measure based on which the clustering is decided. This approach improves the approach of Ziadi (Ziadi

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

et al., 2012) since it extracts mandatory features and optional features along with some constraints among features like And and Require. However, it does not treat product variants wit different structures or different terminologies.

(Salman et al., 2012) present a genetic algorithm to recover traceability links between feature models and source code. Traceability links in SPL are needed to relate variation points and variants with all corresponding low level artifacts (requirements, design, source code and test cases artifacts). The genetic algorithm can determine approximately the implementation of each feature (by linking the feature to classes). However, it generates just one solution for each run, and the number of runs necessary to determine all possible classes for each feature is unknown. In addition, when the number of features and classes grows, the number of possible implementations for each variable feature grows exponentially.

In summary, considering the semantic aspect when extracting the SPL from product variants source code is missing in existing works. In addition, existing approaches suppose that the product variants have a similar/identical structure and the same terminology. However, even though products may vary in their package, class, attribute, and method names and structures, they solve semantically the same problems in their domain. These semantic relationships must be accounted for.

3 FEATURE MODEL IDENTIFICATION BASED AND SEMANTIC CRITERIA

As argued in Section 2.2, considering only elements with identical names and structures when extracting feature models from existing products is too restrictive. Semantics carried through the names of the classes, packages, attributes and method declarations is also important to identify the features and their constraints.

To account for semantics, our feature model extraction operates in three steps (*cf.*, Figure 1):

- Name harmonization: in this pre-processing step, the semantic correspondences among the names of the packages, classes, methods and attribute are treated. This step relies on linguistic and typing information to harmonize the names. It also resolves the problems where the same attribute exists but with different types (*e.g.*, integer or string).

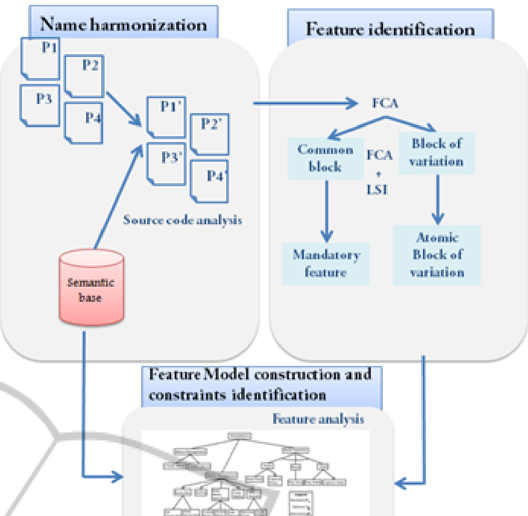


Figure 1: Feature model extraction process.

- Features identification: In order to tolerate some structural differences among the source codes of the product variants, we adapt the FCA (Loesch and Ploedereder, 2007) and LSI (Binkley and Lawrie, 2011) by considering the semantics.
- Feature model construction and constraints identification: In this last step, the semantic information is used first to define the hierarchy in the feature model and, secondly, to extract the types of constraints among identified features. For this, a set of semantic criteria are used to ensure that all constraints among the features are extracted correctly.

In order to determine the correspondences between the names, we adapted the set of semantic criteria we defined in our previous works on constructing frameworks (Ben-Abdallah et al., 2004). The following three criteria express linguistic relationships between element names (however, the list can be extended):

- Synonyms(C1,···,Cn): implies that the names are either identical or synonym, *e.g.*, Mobile-Mobile and Phone-Mobile.
- Hypernyms(C1; C2,···,Cn): implies the name C1 is a generalization of the specific names C2 ,···, Cn, *e.g.*, Media-Video.
- Meronyms(C1; C2): implies that the name C1 is a string extension of the name of the class C2, *e.g.*, Image-NameImage.

The determination of the above linguistic/semantic relationships can be handled through either a dictionary (*e.g.*, Wordnet), or a domain ontology when available.

We will illustrate the steps of our approach through the example of an SPL for mobile phones. This SPL example is a software product family with nine product variants Our approach takes as input the source code of a set of these product variants. Each product implements a simple mobile application.

3.1 Name Harmonization

The product variants must be first harmonized to become suitable as input to the LSI (in step 2). This preprocessing starts by identifying the semantic correspondences between the names of packages, classes, methods and attributes names. The semantic relations are examined in the following order: the equivalence (Synonyms), the string extension (Meronyms), and then the generlization (Hypernyms).

All element names identified as equivalent are harmonized by keeping *one* version of them across all the product variants. In our mobile phone, in the class "AddPhotoToAlbum", we found "GetPhotoName()" and "SetPhotoName()"; these two methods are equivalent. A second example is the methods "GetItemName()" and "SetItemName()" are also equivalent. A third example is the equivalence between the attributes "image" and "photo". For each pair of equivalent names, one of them is chosen to replace all other occurrences of the name.

After the equivalent names are harmonized, the pre-processing step considers the semantic relationship "Meronyms(C1; C2)" to replace the occurrences of C2 by C1 which, being an extension of C2, is more informative. In our example, we found "Image" and "ImageName". At this stage, "ImageName" will replace all occurrences of "Image" in all product variants.

Finally, in general and especially with JAVA, the class and its constructor has the same name. In this case, we ignore one of them to save time. For example, in the class AlbumListScreen, we found methods having the same name AlbumListScreen(), AlbumListScreen(String args0, int arg1), AlbumListScreen(String args0, int arg1, String[] args2, Image[] arg3).

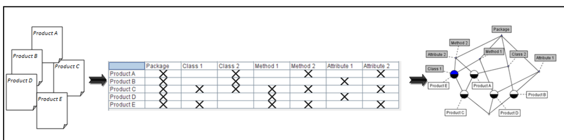
At the end of the pre-processing step, all semantically related names would be harmonized and can then be analyzed through the FCA in the features identification step.

3.2 Features Identification Step

In this step, we use FCA and LSI to extract the commonalities and variabilities among the harmonized product variants. Before explaining this step, let us first overview the basics of FCA and LSI.

Formal concept analysis (FCA) (Ganter and Wille, 1996) is a method of data analysis with a growing popularity across various domains. As illustrated in Figure 2, the main idea of FCA is to analyze data described through the relationships among a particular set of data elements (the table in Figure 2). In our approach, the data represent the product variants being analyzed; the data description is represented through a table where the product variants constitute the rows while source code elements (packages, classes, methods, attributes) constitute the columns of the table. Due to space limitations, Figure 4 shows an extract of this table.

From the table, a concept lattice is derived. The concept lattice permits, in the first time, to define commonalities and variations among all products. (The last part of Figure 2). The top element of the lattice indicates that certain objects have elements in common (i.e., common elements), while the bottom element of the lattice shows that certain attributes fit to common objects (variations). The elements are grouped in blocks. First, common elements are common block which are commonly used in all products. Secondly, the blocks of variations only appear in specific products. Common blocks and block variations are composed of atomic blocks of variation representing only one feature.



	Package	Class 1	Class 2	Method 1	Method 2	Attribute 1	Attribute 2
Product A	X		X		X		X
Product B	X		X			X	
Product C	X	X	X	X	X		X
Product D	X			X		X	
Product E	X	X		X	X		X

Figure 2: Basics of the FCA analysis process.

In our running example, the common block contains all the source code elements that appear in every product. The source code elements that are shared by more than one product are a block of variations. Example,"Package:screen", "Class: AlbumListScreeb", "Class: PhotoViewScreen" presents the common block. Others, like "Class:AddListToAlbum", "Method:SelectionTypeOfMedia" are blocks of variation.

Besides the blocks, the lattice also indicates the relationships among elements. The following relationships can be automatically derived from the sparse representation of the lattice and presented to the analyst:

- *Mandatory*: features appearing at the top concept in the lattice are used in every product.

	Package(Name:Screens)	Class(Name:AddPhotoToAlbum)	Class(Name:NewAlbumScreen)	Class(Name:PhotoAlbumScreen)	Class(Name:PhotoListScreen)	Class(Name:PhotoViewScreen)	Class(Name:SplashScreen)	Class(Name:NewLabelScreen)	Class(Name:AddMediaToAlbum)	Class(Name:MediaListScreen)	Class(Name:PlayMediaScreen)	Class(Name:SelectTypeOfMedia)	Class(Name:CaptureVideoScreen)	Class(Name:PlayVideoScreen)	Class(Name:PasswordScreen)	Class(Name:PasswordListener)	Method(Name:PhotoListScreen)	Method(Name:GetPhotoName)	Method(Name:SetVisibleVideo)	Method(Name:GetCaptureMedia)
Mobile Media 1	x	x	x	x	x	x	x										x			
Mobile Media 2	x	x	x		x	x	x	x									x			
Mobile Media 3	x	x	x		x	x	x	x									x	x		
Mobile Media 4	x	x	x		x	x	x	x									x	x		
Mobile Media 5	x	x	x		x	x	x	x									x	x		
Mobile Media 6	x		x			x	x	x	x	x	x	x								
Mobile Media 7	x		x			x		x	x	x	x	x	x	x					x	x
Mobile Media 8	x		x			x		x	x	x	x	x	x	x					x	x
Mobile Media 9	x		x			x		x	x	x	x	x	x	x	x	x			x	x

Figure 4: Part of the formal context describing mobile systems by source code elements.

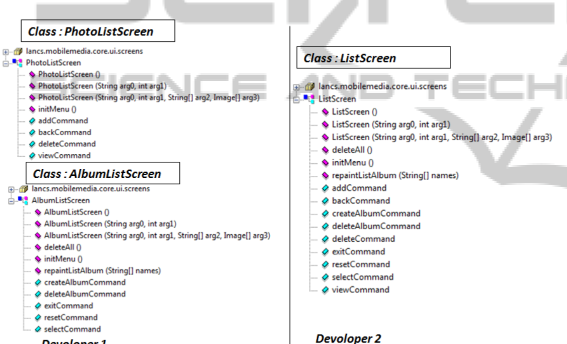


Figure 3: Example of different structures.

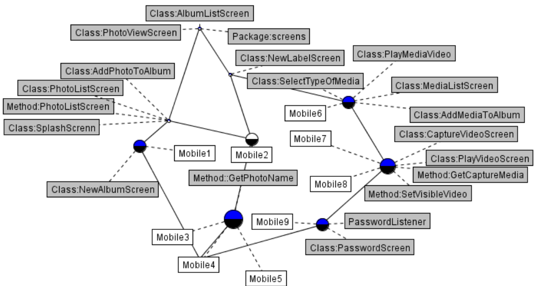


Figure 5: The lattice for the formal context of Table 1.

- *Optional*: features appearing at the bottom concept in the lattice are used in some product.
- *Xor*: Two variable features F1 and F2 that appear in different concepts and whose infimum is the bottom concept are only used in alternative in the product. These features are likely to be Xor features,
- *Require*: if any element has all attributes in F1 also has all attributes in F2
- *AND*: Two features F1 and F2 that appear in the same concept

Recall that, in our work, we suppose that the product variants are implemented by different developers. Consequently, the products may have different structures. For example, a class in one product can be replaced in a second product with two classes where the attributes and methods of the original class are distributed.

In the extract shown in figure 5, we note that the common elements are indicated with the same color. We have common methods and attributes present in two classes for the first developer and in one class for the second developer. Thus, if we consider the owner of the attributes and methods (as treated in (Ziadi et al., 2012)(Al-Msie'Deen et al., 2012)), then these attributes and methods will be considered as different. For example, the owner of method "InitMenu()" is the class "AlbumListScreen" to the first developer (InitMenu(), AlbumListScreen) and the class "ListScreen" to the second developpper (InitMenu(), ListScreen), these pairs are considered different. To resolve the problem of structure, we omitted the owner information.

Moreover, since all the products belong to the same application domain, there are semantic relationships among the words used in the names. To define the similarity, we apply LSI. It allows to measure the similarity degree between names for packages, classes, methods and attributes. Informally, LSI assumes that words that always appear together are related (Binkley and Lawrie, 2011). Consequently, we use LSI and FCA to identify features based on

the textual similarity. Similarity between lines is described by a similarity matrix where the columns and rows represent lines vectors. LSI uses each line in the block of variations as a query to retrieve all lines similar to it, according to a cosine similarity. In our work, we consider the most widely used threshold for cosine similarity that is equals to 0.70 (Binkley and Lawrie, 2011). The similarity matrix which is the LSI result is used as input for the FCA to group the similar elements together based on the lexical similarity. Thus, any document that has similarity with only itself will be ignored. We take the interchanged context as input for FCA which identifies the meaningful groupings of objects that have common attributes.

In our case, each line in the block of variations represents one product variant and at the same time it represents a query. The application of the name harmonization step followed by the identification of feature step extract candidate features without any structure or hierarchy.

The next step in our approach determines the hierarchy and constraints among features and finalizes the feature model construction.

3.3 Feature Model Construction and Constraints Identification

This phase has a threefold motivation. First, the features which are composed of many elements (package, classes, attributes, methods) are renamed based on the frequency of the names of its elements. For example, the feature F1 which is composed of ”Class: AddMediaToAlbum”, ”Method: MediaListScreen”, ”Method: PlayMediaScreen”, ”Method: SelectTypeOfMedia” is renamed MEDIA. Since, it is the word the most frequent and significant.

In addition, the organization and structure of the features is also retrieved based on the semantic criteria. In fact, since the owner information was omitted, then to retrieve the organization of the features, we use the semantic criterion

Hypernyms(FeatureN1, FeatureN2) $\longrightarrow$ *FeatureN1 is the parent of FeatureN2*

For example,the relation Hypernyms(Media, Photo)and Hypernyms(Media, Video) implies that the feature Media is the parent of the features Photo and Video. Moreover, the relation between these feature ”Photo” and ”Video” can be ”OR” or ”XOR”. To resolve this problem, we use Meronyms(Name1, Name2).

Meronyms(FeatureN1, FeatureN2) $\longrightarrow$ *FeatureN1 OR FeatureN2*

In fact, Meronyms(MediaListScreen, PhotoListScreen) and Meronyms(MediaListScreen, VideoListScreen) implies that features Photo and Video are related with OR.

Finally, the constraints between the different features that are extracted with FCA and LSI are verified and some others are added based on the semantic criteria.

Synonyms(FeatureN1, FeatureN2) $\longrightarrow$ *FeatureN1 XOR FeatureN2*

In our running example, we found that the features ”NewAlbumScreen” and ”NewLabelScreen” have equivalent names. Synonyms(NewAlbumScreen,NewLabelScreen) implies that these two features play the same role (Thus one of them is sufficient) and they are related by XOR. However, when we found these two features do not the same parent, therefore they are related with an Exclude relation.

At the end of this last step, all the features are collected in a feature model to specify the variations between these products. The feature model of the mobile system is shown in Figure 6; the features with white circles are Optional while the features with black circles are Mandatory.

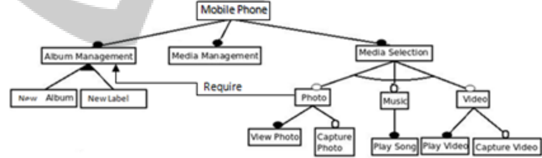


Figure 6: Mobile media system FM.

4 CONCLUSIONS

This paper first overviewed existing works for feature model extraction from product variants. Secondly, it presented a new approach based on a set of linguistic criteria to identify a feature model from different product source codes. Besides accounting for naming differences, our approach has the advantage of identifying automatically the features and their constraints in source codes with different structures. The paper illustrated the proposed approach through the extraction of the feature model of an SPL for mobile phones.

In our ongoing works, we are examining how to add more intelligence in the feature model extraction by considering product variants where the variability is in the body of the operations. We will also consider the use of semantics in the refactoring of software product lines.

REFERENCES

Acher, M., Baudry, B., Heymans, P., Cleve, A., and Hainaut, J.-L. (2013). Support for reverse engineering and maintaining feature models. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 1–8, New York, NY, USA.

Acher, M., Collet, P., Lahire, P., Moisan, S., and Rigault, J. (2010). Modeling variability from requirements to runtime.

Al-Msie'Deen, R., Seriai, A., Huchard, M., Urtado, C., Vauttier, S., and Salman, H. (2012). An approach to recover feature models from object-oriented source code. In *Day Product Line 2012*.

Ben-Abdallah, H., Bouassida, N., Gargouri, F., and Hamadou, A. B. (2004). A uml based framework design method. *Journal of Object Technology*, pages 97–120.

Binkley, D. and Lawrie, D. (2011). Information retrieval applications in software maintenance and evolution. *In Encyclopedia of Software Engineering*, pages 454–43.

Clements, P. and Northrop, L. (2001). Software product lines: Practices and patterns. *SEI Series in Software Engineering*.

Couto, M., Valente, M., and Figueiredo, F. (2011). Extracting software product lines: A case study using conditional compilation. pages 191–200.

Czarnecki, K. and Eisenecker, U. (2000). *Generative programming - methods, tools and applications*. Addison-Wesley.

Ganter, B. and Wille, R. (1996). Formal concept analysis: Mathematical foundations. *Springer-Verlag*.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (foda) feasibility study,. *Technical report CMU/SEI-90-TR-21, Software Engineering Institute,Carnegie Mellon University,*.

Loesch, F. and Ploedereder, E. (2007). Restructuring variability in software product lines using concept analysis of product configurations. pages 159–170.

Lozano, A. (2011). An overview of techniques for detecting software variability concepts in source code. In *ER Workshops*, pages 141–150.

Paskevicius, P., Damasevicius, R., and tuikys, V. (2012). Quality-oriented product line modeling using feature diagrams and preference logic. In *Information and Software Technologies*, pages 241–254.

Riebisch, P. (2003). Using feature modeling for program comprehension and software architecture recovery. Huntsville Alabama, USA.

Rubin, J. and Chechik, M. (2012). Combining related products into product lines. In *FASE*, pages 285–300.

Salman, H., Seriai, A., Dony, C., and Al-Msie'Deen, R. (2012). Genetic algorithms as recovering traceability links method between feature models and source code of product variants. In *Day Product Line 2012*.

She, S., Lotufo, R., Berger, T., Wsowski, A., and Czarnecki, K. (2011). Reverse engineering feature models. pages 461–470.

Xue, Y. (2011). Reengineering legacy software products into software product line based on automatic variability analysis. pages 1114–1117.

Ziadi, T. (Decembre, 2004). Manipulation de lignes de produits en uml. *These de doctorat, Universite de Rennes 1*.

Ziadi, T., Frias, L., da Silva, M. A. A., and Ziane, M. (2012). Feature identification from the source code of product variants. pages 417–422.