Feature model recovery from product variants based on a cloning technique

Jihen Maâzoun  
Mir@cl Laboratory  
University of Sfax, Institut Supérieur d'Informatique et de Multimdia  
jihenmaazoun@gmail.com

Nadia Bouassida  
Mir@cl Laboratory  
University of Sfax, Institut Supérieur d'Informatique et de Multimdia  
Nadia.Bouassida@isimsf.rnu.tn

Hanène Ben-Abdallah  
King Abdulaziz University, Jeddah, Kingdom of Saudi Arabia  
HBenAbdallah@kau.edu.sa

Abstract

A great number of Software Product Lines are not constructed from scratch, they are rather re-engineered from several similar product variants that have been in use. Existing methods for SPL feature model development from product variants may not be applicable when these products are developed independently and hence differ in their element names, methods code and structures. When different developers were involved in the development of the product variants, the naming assumption becomes too difficult to meet. In addition, the variants’ code structures are often different when developed separately and/or when one variant is derived from another through several modifications. Furthermore, while an operation may keep its signature and name in different code variants, its internal code may be different to meet the specifics of each product variant. This paper tackles these three differences by proposing a feature model extraction method that harmonizes the names in the product variants using semantic criteria, tolerates structural differences, and identifies code variability through a clone-code detection technique. It illustrates the method applicability through an example and a CASE tool automating the method’s steps.

1 INTRODUCTION

Software Product Lines (SPLs) [5] have been recognized as a technique that improves productivity in software development. Using an SPL to derive a software product both accelerates the development process and ensures the derivation of a good quality product. The features of an SPL can be used in different combinations to derive product variants in the application domain of the SPL. A feature is a prominent or distinctive quality or characteristic of a software system or systems [7]. An SPL is usually characterized by a feature model (FM) that indicates the features and their combination constraints (and, or, require,…).

In practice, the development of an SPL rarely starts from scratch. An SPL is often setup after several “similar” products have been developed usually via copy & paste techniques. This practical fact was used by several researchers to develop methods for the extraction of SPLs and their feature models from the source code of existing product variants, cf., [1, 14]. These proposed methods are highly dependent on assumptions stemming from the use of copy & paste techniques. More specifically, they rely on three hypotheses about the product variants: they use the same vocabulary to name elements in their source code, they have very similar/identical structures, and the product variability which is encapsulated in the body of their operations is unimportant.

The “names-same structure” assumptions may not hold in the general setting where an SPL should be constructed from product variants that were produced by different developers, and/or product variants that endured too many modifications. In fact, 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 difficult to meet. In addition, the variants’ code structures are often different when developed separately and/or when one variant is derived from an-
other through several modifications. 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 of structural differences, existing feature identification methods would fail. Furthermore, while an operation may keep its signature and name in different product variants, its internal code may be different in order to meet the specifics of each product variant. If the degree of coding differences is very high, such an operation should not be identified as a core feature of the SPL because the product variants actually have different operations. Existing feature model identification methods do not treat the method’s level of detail.

In our approach, we take into account the difference between the bodies of methods through code cloning. Many studies exist on code clones in object-oriented software systems. Some of them only focus on whether code clones exist or not [3] whereas others analyze code clones with respect to their effects, their removal or other peculiarities, identifying crosscutting concerns [2][11].

In this paper, we propose a bottom-up SPL construction approach that both accounts for the differences in the names and structures of the product source codes, and identifies automatically the feature constraints along with the features. The aim of this paper is to complete our approach [10] that extracts the feature model from product source codes and to present its associated tool.

More specifically, compared to our previous work [10], this paper has two contributions. First, it augments our feature model extraction method by the identification of method codes cloning. Secondly, it presents a CASE tool, named FMr-T (Feature Model recovery Tool), that implements our method.

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 method for feature extraction using the semantics enriched with method code clone detection. Section 4 describes the overall architecture of the tool and illustrates its functionality through an example of an SPL for banking systems. Finally, section 5 summarizes the paper and outlines our future work.

## 2 RELATED WORK

An SPL is often modeled in terms of a feature model. Kang et al [7] first proposed the use of feature models in 1990 as part of the Feature Oriented Domain Analysis (FODA). A feature can be seen as a system property or functionality that is relevant to some stakeholders. It is used to capture commonalities or differences among products in an SPL. In fact, feature models are used to specify members of a product-line.

A feature model is a hierarchically arranged set of features. Relationships between a parent (or compound) feature and its child features (or subfeatures) designate the following selection strategies among features when deriving a product in the line of the SPL:

- Mandatory: (sub)features that must be present in every product in the line.
- Optional: (sub)features that may be present in some products.
- And: all subfeatures must be selected together.
- Xor: only one subfeature can be selected.
- Or: one or more subfeature(s) can be selected.
- Require: the selection of one subfeature necessitates the selection of the other.
- Exclude: two subfeatures cannot be part of the same product.

Note that a feature can be either simple/elementary like a package and a class, or composed of several elements like \{package, Class\}, \{package, Class, attribute, method\}...

Several works investigated feature model extraction from the source code of products in order to construct the SPL ([14], [1],[12], [9], [13]). For instance, Ziadi [14] 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. The obtained results show that the approach can not handle products with variable names of classes, methods and attributes. Moreover, this approach does not examine the body of the methods.

On the other hand, Al-Msie’Deeen [1] 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 [14] 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 with different structures or different terminologies.

In summary, the above reviewed works ignore the semantic aspect when extracting the SPL from the source code...
of product variants. Besides the same terminology hypothesis, they suppose that the product variants have a similar/identical structure. 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 in spite of the structural differences.

3 SEMANTICS-BASED FEATURE MODEL EXTRACTION

In our approach, we suppose that products variants may have been constructed by different developers. Hence, the names and structures used may be different. Our approach exploits the semantics carried through the names of the classes, packages, attributes and method declarations within the product variants. In the other hand, it resolve the problems where the same method exists but with different bodies.

To account for the semantics, our feature model extraction operates in the following three steps:

• Name harmonization: in this pre-processing step, the semantic correspondences among the names of the packages, classes, methods and attributes are treated. This step relies on linguistic and typing information to harmonize the names and content of the method. It uses a cloning technique to resolve the problems where the same method exists but with different bodies.

• Features identification: In order to tolerate some structural differences among the source codes of the product variants, we adapt the FCA [8] and LSI [4] 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.

3.1 Name harmonization

This pre-processing starts by identifying the semantic correspondences between the names of packages, classes, methods and attributes. For this step, we adapted the set of semantic criteria we defined in our previous works on constructing frameworks. 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.

• Str_extension(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.

The above semantic criteria are insufficient when two methods have the same or synonymous names but different bodies. To resolve this problem and to extract the variability encapsulated in the body of the methods, we adapt a code cloning technique. To determine code cloning, we propose the following three rules:

<table>
<thead>
<tr>
<th>Rule1</th>
<th>Method names are synonyms but bodies are different.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Synonyms(Mp1,Mp2)) and (Body(Mp1)= Body(Mp2))</td>
<td>→ Method names are kept unchanged</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule2</th>
<th>Method names and bodies are synonyms.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Synonyms(Mp1,Mp2)) and (Body(Mp1)=Body(Mp2))</td>
<td>→ one of the method names must be changed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule3</th>
<th>Same Method names but method bodies are different.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Mp1 =Mp2) and (Body(Mp1) = Body(Mp2))</td>
<td>→ One of the method names must be changed</td>
</tr>
</tbody>
</table>

Note that, in example 1 of Figure 1, the methods Correction() and Change() are synonyms but they have different bodies. The difference between the bodies of these two methods exist in the loop: the method Correction() contains a for loop nested whereas the method Change() contains the while loop. Since, these methods have synonyms names and different structures, thus, there is no need to change their names. This case is treated by Rule 1.

In example 2 of Figure 1, the methods PointerPressed() and PointerPressed(int x, int y) have the same name but the bodies are different: the second method PointerPressed(int x, int y) differs from the first in the conditional structure if. In this case, one of the methods’ names must be changed as proposed in Rule 3.

In example 3 of Figure 1, the methods add_photo() and AddPhoto() are synonyms and they have the same code. In this case, both names are transformed to a unique name as proposed in Rule 2.

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 Feature identification step

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

Formal concept analysis (FCA) [6] is a method of data analysis with a growing popularity across various domains. FCA permits to analyze data described through the relationships among a particular set of data elements. 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.

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 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 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.

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 that are used in every product and that appear at the top of the concept lattice.
- **Optional**: features that are used only in some products and that appear at the bottom of the concept lattice.
- **Xor**: 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 (package, class, method, attribute) belonging to the feature F1 has in common attributes and methods with F2. Then they will be related with a require.
- **AND**: features F1 and F2 that appear in the same concept.

LSI measures the similarity degree between names of packages, classes, methods and attributes. Informally, LSI assumes that words that always appear together are related [4]. 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 equals 0.70 [4]. 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, only with itself will be ignored. We take the interchanged context as input for FCA which identifies the meaningful groupings of objects that have common attributes.

The application of the name harmonization step followed by the feature identification step extracts candidate features without any structure or hierarchy. The next step in our

---

**Figure 1. Difference between bodies of two methods**

```java
// Method 1
public void Correction() {
    DOMParser parser = new DOMParser();
    parser.parse("fichier.xml");
    try {
        parser.parse("fichier.xml");
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

```java
// Method 2
public void Modification() {
    DOMParser parser = new DOMParser();
    parser.parse("fichier.xml");
    parser.parse("fichier.xml");
    for (Element element : listElement) {
        if (element.getAttribute("listElement")
            String s = element.getAttributeText()
        )
            System.out.println(s);
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

---

**Figure 1. Difference between bodies of two methods**

```java
// Method 1
public void PointerPressed(int x, int y) {
    if (x < y) {
        dismiss();
    }
}
```

```java
// Method 2
public void PointerPressed() {
    dismiss();
}
```

```java
// Method 1
public String Add() {
    return image.getString();
}
```

```java
// Method 2
public String Add_Photo() {
    return image.getString();
}
```
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. 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) \(\rightarrow\) FeatureN1 is the parent of FeatureN2
- str_extension(FeatureN1, FeatureN2) \(\rightarrow\) FeatureN1 OR FeatureN2
- Synonyms(FeatureN1, FeatureN2) \(\rightarrow\) FeatureN1 XOR FeatureN2

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

As an example, the constraints between the names (package, class, method, attribute) are adapted the set of semantic criteria defined in Wordnet (see interface 1 of figure 2). The semantic relationships between the names of methods, classes, attributes, and packages of two sources of product lines codes are presented (synonymy, hypernymy, meronymy and str_extension (defined in section 3.1)) After the semantic verification, the clone detection is performed. This allows to check the methods’ bodies and detect some structural differences. In fact, a clone detection strategy is specified to verify the methods bodies using the rules presented in section 3.2. In our example, the interface 2 of Figure 2, presents the GUI to visualize variability in the body of methods. Methods sum() and account() are synonyms. In fact, we display the “clone detection” button that permits to apply rules presented in section 3.2 and display the result of clone detection as shown in part “result” of interface 2 of Figure 2. As result, the method sum() and account() will not be changed.

After applying the name harmonization, the XML file will be updated and will be an input of FCA method (see the interface 3 of figure 2) that permits to determine common blocks and blocks of variation (shown in the interface 4 of figure 2).

Common blocks and blocks of variations are composed of atomic blocks of variation representing only one feature. To define features, we apply LSI with Matlab. According to a cosine similarity that is equal to 0.70, LSI uses each line in the block of variations as a query to retrieve all lines similar to it. 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.

4.2 The functionalities of FMr-T

To illustrate the functionalities of the FMr tool, let us consider a set of banking software products. These source codes are developed with the JAVA language. In the first step, the user chooses the source code file, then the tree will be extracted and saved in an XML document. The XML document corresponds to the name of elements of the parsed code (package, class, method, attribute).

In the second step, to determine the correspondences between the names (package, class, method, attribute), we adapted the set of semantic criteria defined in Wordnet (see interface 1 of figure 2). The semantic relationships between the names of methods, classes, attributes, and packages of two sources of product lines codes are presented (synonymy, hypernymy, meronymy and str_extension (defined in section 3.1)) After the semantic verification, the clone detection is performed. This allows to check the methods’ bodies and detect some structural differences. In fact, a clone detection strategy is specified to verify the methods bodies using the rules presented in section 3.2. In our example, the interface 2 of Figure 2, presents the GUI to visualize variability in the body of methods. Methods sum() and account() are synonyms. In fact, we display the “clone detection” button that permits to apply rules presented in section 3.2 and display the result of clone detection as shown in part “result” of interface 2 of Figure 2. As result, the method sum() and account() will not be changed.

After applying the name harmonization, the XML file will be updated and will be an input of FCA method (see the interface 3 of figure 2) that permits to determine common blocks and blocks of variation (shown in the interface 4 of figure 2).

Common blocks and blocks of variations are composed of atomic blocks of variation representing only one feature. To define features, we apply LSI with Matlab. According to a cosine similarity that is equal to 0.70, LSI uses each line in the block of variations as a query to retrieve all lines similar to it. 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.

5 CONCLUSION

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 and it resolves the problem of variability in methods bodies. The paper illustrated the proposed approach.
through the extraction of the feature model of an SPL for banking systems. The case study was conducted through the "FMr-T" tool which implements the steps of our approach.

In our ongoing works, we are examining how to add more intelligence in the feature model extraction. We will also consider the use of semantics in the refactoring of software product lines.

References


