Feature location in a collection of software product variants using formal concept analysis

Ra'Fat Ahmad Al-Msie'Deen, Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, Hamzeh Eyal-Salman

Feature Location in a Collection of Software Product Variants Using Formal Concept Analysis

Ra'Fat AL-msie'deen[1,⋆], Abdelhak Seriai[1], Marianne Huchard[1], Christelle Urtado[2], Sylvain Vauttier[2], and Hamzeh Eyal Salman[1]

[1] LIRMM / CNRS & Montpellier 2 University, France
{Al-msiedee,Seriai,huchard,eyalsalman}@lirmm.fr
[2] LGI2P / Ecole des Mines d'Alès, Nîmes, France
{Christelle.Urtado,Sylvain.Vauttier}@mines-ales.fr

Abstract. Formal Concept Analysis (FCA) is a theoretical framework which structures a set of objects described by properties. In order to migrate software product variants which are considered similar into a product line, it is essential to identify the common and the optional features between the software product variants. In this paper, we present an approach for feature location in a collection of software product variants based on FCA. In order to validate our approach we applied it on a case study based on ArgoUML. The results of this evaluation showed that all of the features were identified.

Keywords: Software Product Variants, Feature Location, FCA.

1 Introduction

Software product variants often evolve from an initial product developed for and successfully used by the first customer. These product variants usually share some common features but they are also different from one another due to subsequent customization to meet specific requirements of different customers [1]. As the number of features and the number of product variants grows, it is worth reengineering product variants into a Software Product Line (SPL) for systematic reuse. To switch to Software Product Line Engineering (SPLE) starting from a collection of existing variants, the first step is to mine a feature model that describes the SPL. This further implies to identify the software family's common and variable features. Manual reverse engineering of the feature model for the existing software variants is time-consuming, error-prone, and requires substantial effort [2]. Thus, we propose in this paper a new approach for feature location in a collection of software product variants. Our approach is based on the identification of the implementation of these features among object-oriented (OO) elements of the source code. These OO elements constitute the initial search space. We rely on Formal Concept Analysis (FCA) to reduce this search

⋆ This work has been funded by grant ANR 2010 BLAN 021902.

J. Favaro and M. Morisio (Eds.): ICSR 2013, LNCS 7925, pp. 302–307, 2013.

space by identifying maximal subsets of features shared by maximal subsets of product variants and organizing these subsets by inclusion.

Our approach is detailed in the remainder of this paper as follows. Section 2 presents FCA, and Section 3 outlines our approach. Section 4 discusses our implementation and evaluation. Section 5 presents the related work. We conclude and draw perspectives for this work in Section 6.

2 Formal Concept Analysis (FCA)

Galois lattices and concept lattices [3] are core structures of a data analysis framework (FCA) for extracting an ordered set of concepts from a dataset, called a formal context, composed of objects described by attributes. A formal context is a triple $K = (O, A, R)$ where O and A are sets (objects and attributes, respectively) and R is a binary relation, i.e., $R \subseteq O \times A$. An example of formal context is provided in Figure 1 (left). A formal concept is a pair (E, I) composed of an object set $E \subseteq O$ and its shared attribute set $I \subseteq A$. $E = \{o \in O | \forall a \in I, (o, a) \in R\}$ is the extent of the concept, while $I = \{a \in A | \forall o \in E, (o, a) \in R\}$ is the intent of the concept. Given a formal context $K = (O, A, R)$, and two formal concepts $C_1 = (E_1, I_1)$ and $C_2 = (E_2, I_2)$ of K, the concept specialization order $\leq_s$ is defined by $C_1 \leq_s C_2$ if and only if $E_1 \subseteq E_2$ (and equivalently $I_2 \subseteq I_1$). C_1 is called a sub-concept of C_2. C_2 is called a super-concept of C_1. Let $\mathcal{C}_K$ be the set of all concepts of a formal context K. This set of concepts provided with the specialization order $(\mathcal{C}_K, \leq_s)$ has a lattice structure, and is called the concept lattice associated with K. In our approach, we will consider the AOC-poset (for Attribute-Object-Concept poset), which is the sub-order of $(\mathcal{C}_K, \leq_s)$ restricted to object-concepts and attribute-concepts. An object-concept (resp. attribute-concept) is the lowest concept (resp. a greatest concept) where an object (resp. an attribute) appears. In AOC-poset representations, objects are represented only in their introducer concept (and inherited by superconcepts), while attributes are represented only in their introducer concept (and inherited by their subconcepts), meaning that no concept should have empty object part and empty attribute part.

3 Our Approach to Feature Location

This section provides main concepts and hypotheses used in our approach.

3.1 Goal and Core Assumptions

The general objective of our work is to identify a feature model for a collection of software product variants based on the static analysis of their source code. We consider that "a feature is a prominent or distinctive and user visible aspect, quality, or characteristic of a software system or systems" [4]. We adhere to the classification given by [4] which distinguishes three categories of

features: functional, operational and presentation features. Our work focuses on the identifying of functional features. In our approach we deal with software systems where the functional features are implemented at the programming language level (*i.e.*, source code). The functional features are implemented by object oriented building elements (OBEs) such as *packages*, *classes*, *attributes*, *methods* or *method body elements* (*local variable*, *attribute access*, *method invocation*). We consider that a feature corresponds to exactly one set of OBEs. This means that a feature always has the same implementation in all products where it is present. We also consider that feature implementations may overlap: a given OBE can be shared between several features' implementations. In this paper, we name such shared OBE as a *junction*.

3.2 Features versus Object-Oriented Building Elements

Feature location in a collection of software variants consists in identifying a group of OBEs that constitutes its implementation. This group of OBEs must either be present in all variants (case of a common feature) or in some but not all variants (case of an optional feature). Thus, the initial search space for the feature location process is composed of all the subsets of OBEs of existing product variants. As the number of OBEs is big, a strategy must be designed to reduce the search space.

Our proposal consists in dividing the OBE set in specific subsets: the *common feature set* – also called *common block* (CB) – and several *optional feature sets* (Block of Variations, denoted as BVs). Optional (*resp.* common) features appear in some but not all (*resp.* all) variants, they are implemented by OBEs that appear in some but not in all (*resp.* all) variants.

This is realized by building a formal context, which is composed of software variants (objects of the formal context) described by their OBEs (attributes of the formal context). The relation associates a software variant with the OBEs that appear in its source code. The corresponding AOC-poset is then calculated. A concept intent (containing the concept attributes) represents OBEs common to two or more variants (the objects included in the concept extent). As the concepts of the AOC-posets are ordered, the intent of the most general (i.e., top) concept gathers the OBEs that are common to all products. They constitute the CB. The intents of the remaining concepts are BVs. A concept intent corresponds to the implementation of one or more features. As an illustrative example, we consider four text editor software variants. *Editor_1* supports core text editing features: *open*, *close*, and *print* a file. *Editor_2* has the core text editing features and a new *select_all* feature. *Editor_3* supports *copy* and *paste* features, together with the core ones. *Editor_4* supports *select_all*, *copy* and *paste* features, together with the core ones. Figure 1 shows the formal context for the text editor variants and the AOC-poset for this formal context which shows the CB and BVs.

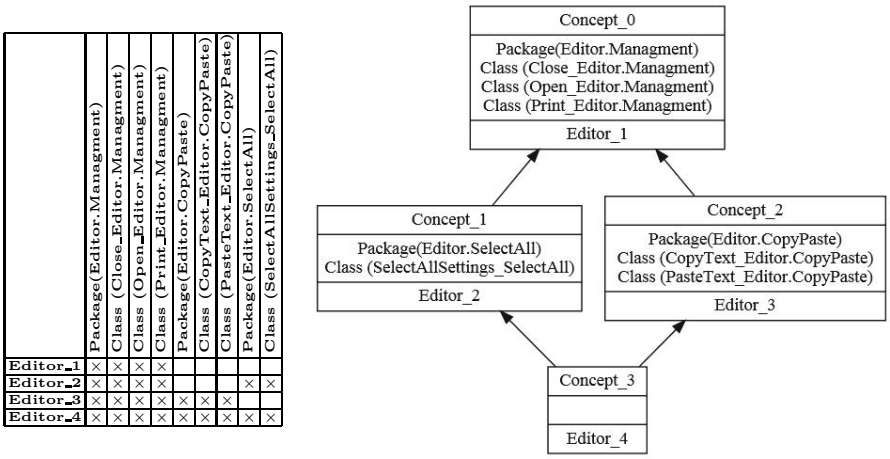

Fig. 1. The Formal Context and AOC-poset for Text Editor Variants

4 Experimentation

To validate our approach, we ran experiments on the Java open-source software ArgoUML [5]. We used 10 variants for ArgoUML. The advantage of ArgoUML variants is that they are well documented and their feature model is available for comparison with our results and validation of our proposal. ArgoUML variants are presented in Table 1: LOC (Lines of Code), NOP (Number of Packages), NOC (Number of Classes) and NOOBE (Number Of Object-oriented Building Elements).

Table 1. ArgoUML software product variants

Product #	Product Description	LOC	NOP	NOC	NOOBE
P1	All features disabled	82,924	55	1,243	74,444
P2	All features enabled	120,348	81	1,666	100,420
P3	Only Logging disabled	118,189	81	1,666	98,988
P4	Only Cognitive disabled	104,029	73	1,451	89,273
P5	Only Sequence diagram disabled	114,969	77	1,608	96,492
P6	Only Use case diagram disabled	117,636	78	1,625	98,468
P7	Only Deployment diagram disabled	117,201	79	1,633	98,323
P8	Only Collaboration diagram disabled	118,769	79	1,647	99,358
P9	Only State diagram disabled	116,431	81	1,631	97,760
P10	Only Activity diagram disabled	118,066	79	1,648	98,777

Table 2 summarizes the obtained results. For readability's sake, we manually associated feature names to CB and BVs, based on the study of the content of each block and on our knowledge of the software. Of course, this does not impact the quality of our results. In Table 2, CB represents a single common feature. For the given set of BVs [2 -10], each BV represents a single optional feature. For given set of BVs [11 - 22], each BV represents a junction between two or more features. The column (# OBEs) in Table 2 represents the number of OBEs that implement this feature.

Table 2. Feature Location in ArgoUML Software Variants

#	Feature Name	# OBEs	#	Feature Name	# OBEs
1	Class Diagram	74431	12	Junction cognitive/deployment	745
2	Diagram	1309	13	Junction cognitive/sequence	55
3	Use case Diagram	1928	14	Junction sequence/collaboration	111
4	Collaboration Diagram	935	15	Junction state/logging	6
5	Cognitive Diagram	10193	16	Junction deployment/logging	18
6	Activity Diagram	1583	17	Junction collaboration/logging	13
7	Deployment Diagram	1334	18	Junction use case/logging	22
8	Sequence Diagram	3708	19	Junction sequence/logging	51
9	State Diagram	2597	20	Junction activity/logging	3
10	Logging	1149	21	Junction cognitive/logging	169
11	Junction activity/state	57	22	Junction between features 14/17/19	18

In fact 22 features have been identified from ArgoUML software product variants. The 12 extra features (features 11-22) represent junctions between the other features [5]. The top concept (feature 1 called "Class Diagram" in Table 2) contains 74431 OBEs that are shared by all software product variants (*i.e.*, CB). In particular, it contains the *class diagram* feature, which is indeed a common feature, and is therefore present in every product. We compared the obtained CB with the common features of the original feature model [5]. CB corresponds exactly to one common feature (*i.e.*, class diagram). Concerning the obtained BVs, each BV from 2-10 corresponds exactly to one original optional feature. For BVs from 11-22, each block represents a junction.

5 Related Work

Loesch et al. [6] applied FCA to analyze the variability in a software product line based on product configurations (described by features), and construct a lattice that provides a classification of the usage of variable features in real products derived from the product line. An inclusive survey about approaches linking features and source code in a single software is proposed in [7]. Rubin *et al.* [8] present an approach to locate optional features from two product variants' source code. They do not consider common features and limit their proposal to two variants. Xue *et al.* [1] propose an automatic approach to identify the traceability links between a given collection of features and a given collection of source code variants. They thus consider feature descriptions as an input. Acher *et al.* [2] present automated techniques to extract variability descriptions in a software architecture and consider the architect's knowledge for reverse engineering architectural feature models. She *et al.* [9] propose an approach to define a feature model based on a set of already identified features. The main problem tackled is to identify the structure of the feature model. In particular, they present procedures to identify alternatives from an existing set of features. Their work is complementary to our work as it can take as input a feature set deduced from our approach and synthesize the feature model. Acher et al. [10] present an approach to synthesize a feature model based on the product descriptions. Their approach takes as input product description for a collection of product variants to build the FM. Products are described by characteristics (language, license, etc.) with different patterns on values (many-valued, one-valued, etc.).

Ryssel et al. [11] applied FCA to extract feature diagrams from an incidence matrix that contains matching relations as input. The matrix shows the parts of a set of function-block oriented models that describe different controllers of a DC motor. The approach proposed by Ziadi *et al.* [12] is the closest one. Authors propose a solution for feature identification from the source code of a set of product variants. They identify all common features as a single mandatory feature. However, they do not distinguish between optional features that appear together in a set of variants. Their approach doesn't consider the method body and do not use any classification technique to classify object oriented elements.

6 Conclusion and Perspectives

We present in this paper an approach for feature location in a collection of software product variants based on FCA. It has been applied on a collection of ArgoUML software products. The results of this evaluation showed that all of the features were identified. As future work, we will apply a clustering algorithm on the CB and BVs to determine more precisely each feature implementation based on both lexical similarity (*i.e.*, textual similarity between OBEs) and semantic similarity/dependency structure (*i.e.*, inheritance, attribute access, method invocation). We also plan to use the identified common and optional features to automate the building of the studied software family's feature model.

References

1. Xue, Y., Xing, Z., Jarzabek, S.: Feature location in a collection of product variants. In: IEEE 19th RE Conference, pp. 145–154 (2012)
2. Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., Lahire, P.: Reverse engineering architectural feature models. In: Crnkovic, I., Gruhn, V., Book, M. (eds.) ECSA 2011. LNCS, vol. 6903, pp. 220–235. Springer, Heidelberg (2011)
3. Ganter, B., Wille, R.: Formal Concept Analysis, Mathematical Foundations. Springer (1999)
4. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study (1990)
5. Couto, M., Valente, M., Figueiredo, E.: Extracting software product lines: A case study using conditional compilation. In: 15th CSMR Conference, pp. 191–200 (2011)
6. Loesch, F., Ploedereder, E.: Optimization of variability in software product lines. In: IEEE 11th ISPL Conference, pp. 151–162 (2007)
7. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process, 53–95 (2012)
8. Rubin, J., Chechik, M.: Locating distinguishing features using diff sets. In: 27th ASE Conference, ASE 2012, pp. 242–245. ACM (2012)
9. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: ICSE, pp. 461–470 (2011)
10. Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C.: On extracting feature models from product descriptions. In: VaMoS, pp. 45–54. ACM (2012)
11. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Extraction of feature models from formal contexts. In: 15th ISPL Conference, pp. 4:1–4:8. ACM (2011)
12. Ziadi, T., Frias, L., da Silva, M.A.A., Ziane, M.: Feature identification from the source code of product variants. In: CSMR 2012, pp. 417–422 (2012)