An Overview of Techniques for Detecting Software Variability Concepts in Source Code

Angela Lozano*

Université catholique de Louvain (UCL), ICTEAM,
Place Sainte Barbe 2, B-1348 Louvain La Neuve, Belgium

Abstract. There are two good reasons for wanting to detect variability concepts in source code: migrating to a product-line development for an existing product, and restructuring a product-line architecture degraded by evolution. Although detecting variability in source code is a common step for the successful adoption of variability-oriented development, there exists no compilation nor comparison of approaches available to attain this task. This paper presents a survey of approaches to detect variability concepts in source code. The survey is organized around variability concepts. For each variability concept there is a list of proposed approaches, and a comparison of these approaches by the investment required (required input), the return obtained (quality of their output), and the technique used. We conclude with a discussion of open issues in the area (variability concepts whose detection has been disregarded, and cost-benefit relation of the approaches).

1 Introduction

Today's companies face the challenge of creating customized and yet affordable products. Therefore, a pervasive goal in industry is maximizing the reuse of common features across products without compromising the tailored nature expected from the products. One way of achieving this goal is to delay customization decisions to a late stage in the production process, which can be attained through software. For instance, a whole range of products can be achieved through a scale production of the same hardware, and a software customization of each type of product.

The capability of building tailored products by customization is called variability. Software variability can be achieved through combination and configuration of generic features.

Typical examples of variability can be found in embedded software and software families. Embedded software facilitates the customization of single purpose machines (i.e. those that are not computers) such as cars, mobile phones, airplanes, medical equipment, televisions, etc. by reusing their hardware while

* Angela Lozano is funded as a post-doc researcher on a an FNRS-FRFC project. This work is supported by the ICT Impulse Program of ISRIB and by the Inter-university Attraction Poles (IAP) Program of BELSPO.

O. De Troyer et al. (Eds.): ER 2011 Workshops, LNCS 6999, pp. 141–150, 2011.

varying their software to obtain different products. Software families are groups of applications that come from the configuration and combination of generic features. Both embedded software and software families refer to groups of applications related by common functionality. However, variability can also be found in single applications when they delay design decisions to late stages in order to react to different environments e.g. mobile applications, games, fault tolerant systems, etc.

Motivations for Detecting Variability: When companies realize that slight modifications of a product could enlarge their range of clients they could migrate to a product-line development. Detecting variability opportunities in the current product is the first step to assess which changes have a higher return in terms of potential clients. This return assessment is crucial for the success of such migration because the reduction on development cost may not cover the increase in maintenance cost if the variation introduced is unnecessary.

Once a product-line architecture is in place, it will degrade over time. Degradation of the architecture is a reality of software development [7]. In particular, product-lines have well-known evolution issues that can degrade their architecture [3,12]. Therefore, at some point it might become necessary to restructure the product-line; and the first step is reconstructing its architecture from the source code.

Both previous scenarios start from the source code of the (software) system as source of information. Source code mining is a reverse engineering technique that extracts high-level concepts from the source code of a system. In order to mine for variability concepts, we need to clearly define these high-level concepts.

Although mining for variability in source code is a common step towards the successful adoption of variability-oriented development, there exists no compilation nor comparison of existing approaches available to attain this task. The goal of our paper is two-fold. First, we identify high-level concepts targeted by variability mining approaches and describe the intuition behind them. Second, we use these concepts to classify the approaches that detect variability as they uncover the intuition and assumptions behind the technique used.

2 Variability Concepts

In order to compare variability mining approaches we establish a common ground of terms found in literature. The purpose of this common ground is not to provide a formal definition of each term, but to fix its meaning in an intuitive manner. In particular, we consider four variability concepts central to variability mining: features, variability dependencies, variation points, and variants.

Features: Variability is described in terms of *features*. There are two types of features: mandatory and variable. To illustrate these kinds of features we present an example of the possible configurations of features in a car. Figure 1 depicts the car feature diagram, illustrating the relations between variability concepts.

A *feature* represents a unit of software functionality i.e. an increment of behavior (w.r.t. some standard car behavior). The features in Figure 1 are the

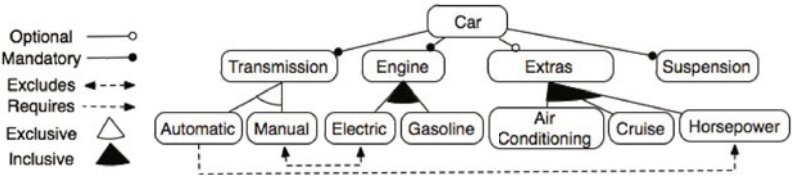

Fig. 1. A feature diagram

suspension, transmission, engine, and extra options. *Mandatory features* can be implemented directly in the application because they do not depend on the usage environment i.e. they do not require customizations. The mandatory features of Figure 1 are the suspension, transmission and engine. Mandatory features represent functionality that must be included in all products.

Variants: As mentioned in the introduction, the goal of variability is to allow for flexibility by delaying customization decisions. *Variable features* are those features that require customizations. The transmission, engine, and extras are the variable features of Figure 1, because they can vary from one car to another. The options available for a variable feature are called its *variants*. The transmission is a variable feature with two variants: automatic and manual (Fig. 1).

Depending on its variants a feature can be classified into optional or alternative [1]. *Optional features* do not require a variant and are included only in certain products, e.g. the extra options in a car (see example on Fig. 1). *Alternative features* implement an obligatory feature in the product family in different ways depending on the product developed. Alternative features are composed of mutually inclusive or mutually exclusive variants [1]. *Mutually inclusive* variants depend on each other, so including one of them requires to include the rest of them. Having mutually inclusive variants turns an alternative feature into a *multiple feature* because it can have several values assigned. *Mutually exclusive* variants are interchangeable. Having mutually exclusive variants make an alternative feature a *single feature* because it can have only one value assigned. In the example of Fig. 1, the transmission is a single feature because the manual and the alternative transmissions are mutually exclusive, while the engine is a multiple feature because the gasoline and electric engines are mutually inclusive.

Variable features can be *single* if they must have one value assigned, *multiple* if they can have several values assigned, and *optional* if they do not require a value assigned. The example shown in Figure 1 presents a single feature (the transmission), a multiple feature (the engine), and an optional feature the extra.

Variability Dependencies: There are two types of *variability dependencies* which dictate constraints among variable features: when including a feature *requires* including another feature, and when including a feature *excludes* or prohibits the inclusion of another feature. For instance, automatic transmission requires extra horsepower while the manual transmission excludes use of an electric engine (Fig. 1).

Variation Points: Variable features are implemented in the application in the form of *variation points*. A *variation point* is the placeholder for a value that has associated variable functionality. Variation points make explicit delayed design decisions. Associating a value to a variation point is called *binding*. Variation points with a value are *bound*, while those without a value are *unbound*.

3 Mining for Feature Diagrams

Feature diagrams describe the commonalities and differences of some domain model. Therefore feature diagram mining requires to analyze multiple applications of the same domain.

Antkiewicz et al. [2] analyzed applications that extend the same framework. These applications can be viewed as instances of the same domain because their usage of the framework shows configurations of the concepts provided by the framework. The authors aim at detecting variable features. The feature to be detected is described by an abstract concept that is the starting point of the analysis. For instance, Applet will look for all instances of Applet in the applications analyzed. A feature is extracted in the form of sets of distinguishing structural and behavioral patterns that allow discriminating configurations of the same concept. The patterns are detected with source code queries. The results of the queries are facts that describe the structural and run-time relations among source code entities. That is, variables/parameters/returned objects and types, methods implemented and understood, calls sent and received, precedence relations between methods called, etc. The mining builds a tree of source code code facts obtained from the source code entity that is the starting point of the analysis. The algorithm then calculates commonalities and differences among the facts to establish mandatory, optional, multiple and indispensable patterns when configuring a feature of the framework.

Yang et al. [24] analyzed open source applications with similar functionality. The assumption is that data models (entity-relationship) of these applications are similar and uncover a basis for mapping the common concepts among different applications. The starting point of the mining is a reference domain data model. The data schema of the applications is then mapped to the reference domain model by detecting entities or fields with similar names in the applications. The approach is based on detecting consistent data access semantics (i.e., similar usage of data entities by the methods in the applications). The data access semantics are obtained by detecting SQL statements using aspects which intercept invocations to the database library and store SQL run-time records. In order to verify that different methods have a similar data access semantics, the SQL records must contain the tables, fields and constraints involved in the queries. These records describe the data access semantics of each method in each application. The records are then analyzed using Formal Concept Analysis (FCA), a classification technique that aims at finding the maximal set of objects that share a maximal set of properties or attributes. The resulting concepts represent the usage of data-entities mapped to the reference domain model. Depending

on the level of the concept, it is merged with neighbor concepts or pruned so that each concept represents one feature. The result is a domain feature diagram comprising mandatory features, variable features, whether they are alternative or multiple, its variants, and whether they are inclusive or exclusive. However, the approach is incapable of detecting variability dependencies.

4 Mining for Variability Dependencies

Variability dependencies are constraints between features that establish the valid products of a feature diagram. In terms of source code, variability dependencies (requires/excludes) are related with control flow relations between features. Nevertheless, other types of feature overlaps have been found by analyzing implementations of features regardless of being variable or mandatory.

Czarnecki et al. [5] define Probabilistic Feature Models for feature diagrams, organized using the probability of having a feature given the presence of another feature. The approach is based on counting the legal configurations of a feature diagram. The legal configurations are used to determine a set of legal samples (of legal configurations), which are in turn used to calculate the frequency of coexistence of all possible combinations of features. These frequencies are then used to obtain conditional probabilities, that is, the probability of requiring a feature given the presence in the application of another feature. Conditional probabilities are used to structure the feature diagram and to decide when a variable feature is optional, alternative, mandatory, inclusive or exclusive. Finally, a Bayesian Network with a Directed Acyclic Graph of the features, and the conditional probability table is used to learn variability dependencies. The Bayesian Network is capable of detecting require relations ($probability(feature1|feature2)$) and exclude relations ($probability(\overline{feature1}|feature2)$) among features. A disadvantage of this approach is that it does not analyze source code but it requires an intermediate approach to detect the Directed Acyclic Graph of features in the domain and their mapping to several products.

Parra et al. [19] propose an analysis of feature constraints based on the assumption that variable features are implemented with aspects. The approach detects that one feature requires a second feature when the pointcut that defines the variation point for the first feature references source code elements referred to by the aspect that defines the second feature. The approach can detect when one feature excludes a second feature, when the pointcuts (that define the variation points) for both features refer to the same source code elements. This analysis is used to detect the correct order (if it exists) to compose the variants represented by aspects in the source code. The disadvantage of this approach is that presupposes an aspect-oriented implementation of variability.

Egyed [6] assumes that calling a similar set of source code entities when executing different features implies that one feature is a sub-feature of the other. Similarly, Antkiewicz et al.[2] consider that a sub-feature is essential if it is common to all applications (of the same domain) analyzed. This approach locates a feature by identifying similar patterns in entities that may be related to the

feature analyzed, a sub set of these patterns is considered a sub-feature. Aside from detecting essential sub-features, their approach is also capable of detecting incorrect features due to missing essential sub-features. However, this approach is incapable of detecting variability dependencies across features that do not have a containment relation.

Lai and Murphy [16] described two situations in which several features are located in the same source code entity: overlap and order. Overlap occurs when there is no control or data flow dependency between the features in the source code entity. Overlap is expected and encouraged in sub-features; however, other types of overlap should be documented and handled during maintenance. Order occurs when the features in the source code entity have a control or data flow dependency, i.e. they require a partial execution order. Usually wherever many features were related, there was no partial order.

In his PhD thesis [11], Jaring proposes four types of variability dependencies depending on the binding of variation points. The categories are: dependencies between variation points , dependencies between a variation point and a variant , dependencies between a variant and a variation point , and dependencies between variants . This characterization is used to uncover hidden variability dependencies in a legacy application (that uses C macros to implement its variability). However it is not clear to what extent the analysis is automated and to what extent it requires user-input.

5 Mining for Variation Points and Variants

Variation points capture the functionality areas in which products of the same product family differ. Mining for variation points aims at detecting source code entities that allow diverging functionality across different products of the same domain. Although there exists literature describing how to implement variability [13,1,4,22,17], we could only find one approach to detect variation points and variants. The lack of approaches may be due to the wide variety of possibilities to translate a conceptual variation point (i.e. a delayed decision) to the implementation of a variation point, as well as to the difficulty to trace this translation [20,14].

Thummalapenta and Xie [23] analyze applications that extend the same framework. They calculate metrics that describe the amount and type of extensions per class and method of the framework. These metrics allow to classify the methods and classes of the framework into variation points (hotspots/hooks) and coldspots/templates. Authors, analyzed the level of variability of the frameworks analyzed by counting the percentage of classes and methods identified as variation points. The disadvantage of this approach is that it requires a high variety of applications from the domain to give reliable results.

A variant is an option available for a variable feature. Mining for variants implies assigning a high level concept to the values (or objects) used in the *variation points* to decide when to change the implementation of a variable feature. This means that mining for variants requires detecting variation points

and linking them to their corresponding variable feature, so it is possible to trace them to a single feature. We could not find any approaches to mine for variants. However this lack of results is predictable because of the lack of approaches to mine for variation points. Given that the approach described above [23] was designed to analyze the flexibility of frameworks, it lacks references to variability, and therefore, to variants.

6 Mining for Products of the Same Domain

Snelting [21] analyzes macros (C's preprocessor directives like *#ifndef*). The lines of code corresponding to each macro are characterized with the global variables (or configuration variables) that affects them. This characterization produces a table where each line has the lines of code of each macro and each column the variables configured in such macro. The table is analyzed using Formal Concept Analysis (FCA). The concepts of the lattice resulting from the analysis represent a possible configuration of the application. The intent of the concepts provides the variables that affect that configuration, and the extent the lines of code affected by that configuration. Moreover, the relations between concepts indicate subsumed configurations. Crossed relations between chains of concepts would indicate interference among those configurations. If the application is implemented separating concerns and anticipating changes the lattice would have disjoint sublattices in the middle of the lattice (i.e. high configuration coupling). Otherwise configuration variables of different features would be merged in single concepts in the middle area of the lattice, indicating dependencies and low coupling.

Hummel et al.[10] analyzed the signatures of the methods of several open source applications to recommend methods that classes in the same domain implement but that are missing from the user's implementation. The approach assumes that the domain concept that a class represents is summarized as the set of signatures of its methods. Using this information, their technique finds all classes with a similar set of signatures to the user's class, and detects the signatures missing from the user implementation. The disadvantage of this technique is that it is limited to the domain concepts that are implemented in open source applications, and therefore it might be difficult to use for business-specific domain concepts.

7 Mining for Variable vs. Mandatory Features

Several approaches mine for features.However, this section focuses on approaches to differentiate between mandatory (common functionality across products of the same domain) and variable features (divergent functionality across products of the same domain). This is usually achieved by analyzing the results of clone detection.

Faust and Verhoef [8] were the first ones to propose the analysis of diverse products of the same domain to explore development paths for mandatory features. They use bonsai maintenance as a metaphor to keep product lines under

control. The approach is called grow and prune because it aims at letting products of the domain evolve (grow), and then merge as mandatory features (prune) the successful source code entities of these products. For that reason, they proposed metrics to evaluate the success of a source code entity. A source code entity is defined as successful if its implementation contained a large amount of code, a low number of decisions, and a low frequency of changes, was highly used and cloned. Nevertheless the approach is incapable of offering further assistance for restructuring the product line.

Mende et al. [18] use clone detection to measure the level of commonalities across directories, files and functions of different products of the same domain. The idea is to assess to what extent a product (p1) can be merged into a basic product (p2), and to what extent the functionality of the product to be merged (p1) is included in the other product (p2). Depending on the number of identical functions (similarity = 1) and of similar functions (similarity between 0 and 1) of the product p2 into the product p1, it is possible to say if the correspondence is identical or potential, and if such correspondence is located in one function or not. Therefore the analysis proposed helps to detect potential mandatory features parts in an existing product line (i.e., product-specific functions identical across several products), as well as variability points that may require some restructuring to separate better mandatory and variable features (i.e., product-specific functions similar across several products).

Frenzel et al.[9] extract the model of a product-line architecture based on several products of the same domain. To extract the model they use reflexion models, which gives them the static components, their interfaces, their dependencies, and their grouping as layers and sub-layers in the system. The model is then compared with the implementation of the products by checking whether different products have corresponding components (based on clone similarity). Clone detection is also used to transfer common implementation areas to the common design. The latter two approaches are merged and further developed in [15]. However, given that these approaches do not mine for features, they are unable infer the feature diagram and its correspondence with the inferred architecture .

8 Conclusions

This paper compiles techniques to propose refactorings from variable to mandatory features, and to improve the comprehension of variable products by discovering the decomposition of features in a domain, the hidden links among implementations of variable features, the source code entities in charge of the variation, and valid products of a domain. Although we found several approaches to mine for most of the variability concepts the approaches can be improved in several ways. Some of the techniques have demanding requirements. For instance, reconstructing feature diagrams require an initial domain model [24] or entity that implements the feature to analyze [2], while detecting *excludes/requires* dependencies may require an aspect-oriented implementation of variable features [19] or an initial feature diagram and its mapping to several applications

[5]. Other techniques have a restricted automated support. For example, reconstructing feature diagrams may require manual post processing [24]. Finally, the usefulness of the output of some techniques is limited to address architectural degradation of product-line design from single products. For instance, knowing the configurations of a product line and the variables involved [21] does not provide any hints on how to restructure it or on how to map this implementation details to domain concepts; which limits the usage of the approach to deal with architectural degradation due to evolution.

Mining for variation points is the area with highest potential because there are several papers that describe how variability should be implemented. Mining for variants also has potential given that is a neglected area, and that it is complimentary to the detection of variation points. Nevertheless, detecting variation points is not enough to detect variants because each variation point needs to be linked to a variable feature. However, the assignment of a variation points to a variable feature could be technically challenging because the majority of order interactions are due to control flow dependencies [16]. Another area open for future work is extending the approaches to mine for feature diagrams, and for variable and mandatory features to analyze the flexibility of single products in order to support the migration towards product-line development.

References

1. Anastasopoulos, M., Gacek, C.: Implementing product line variabilities. In: SSR 2001: Proc. of the 2001 Symposium on Software Reusability, pp. 109–117. ACM, New York (2001)
2. Antkiewicz, M., Bartolomei, T.T., Czarnecki, K.: Fast extraction of high-quality framework-specific models from application code. Autom. Softw. Eng. 16(1), 101–144 (2009)
3. Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., Pohl, K.: Variability issues in software product lines. In: Revised Papers from the 4th Int'l Workshop on Software Product-Family Engineering, PFE 2001, pp. 13–21. Springer, Heidelberg (2002)
4. Brown, T.J., Spence, I., Kilpatrick, P., Crookes, D.: Adaptable components for software product line engineering. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 154–175. Springer, Heidelberg (2002)
5. Czarnecki, K., She, S., Wasowski, A.: Sample spaces and feature models: There and back again. In: SPLC 2008: Proc. of the 2008 12th Int'l Software Product Line Conference, pp. 22–31. IEEE Computer Society, Washington, DC, USA (2008)
6. Egyed, A.: A scenario-driven approach to traceability. In: ICSE 2001: Proc. of the 23rd Int'l Conference on Software Engineering, pp. 123–132. IEEE Computer Society, Washington, DC, USA (2001)
7. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE Trans. Softw. Eng. 27, 1–12 (2001)
8. Faust, D., Verhoef, C.: Software product line migration and deployment. Software: Practice and Experience 33(10), 933–955 (2003)

9. Frenzel, P., Koschke, R., Breu, A.P.J., Angstmann, K.: Extending the reflexion method for consolidating software variants into product lines. In: WCRE 2007: Proc. of the 14th Working Conference on Reverse Engineering, pp. 160–169. IEEE Computer Society, Washington, DC, USA (2007)
10. Hummel, O., Janjic, W., Atkinson, C.: Proposing software design recommendations based on component interface intersecting. In: Proc. of the 2nd Int'l Workshop on Recommendation Systems for Software Engineering, RSSE 2010, pp. 64–68. ACM, New York (2010)
11. Jaring, M.: Variability Engineering as an Integral Part of the Software Product Family Development Process. PhD thesis, Rijksuniversiteit Groningen (2005)
12. Johansson, E., Höst, M.: Tracking degradation in software product lines through measurement of design rule violations. In: Proc. of the 14th Int'l Conference on Software Engineering and Knowledge Engineering, SEKE 2002, pp. 249–254. ACM, New York (2002)
13. Keepence, B., Mannion, M.: Using patterns to model variability in product families. IEEE Softw. 16, 102–108 (1999)
14. Kim, S.D., Her, J.S., Chang, S.H.: A theoretical foundation of variability in component-based development. Inf. Softw. Technol. 47, 663–673 (2005)
15. Koschke, R., Frenzel, P., Breu, A.P., Angstmann, K.: Extending the reflexion method for consolidating software variants into product lines. Software Quality Control 17, 331–366 (2009)
16. Lai, A., Murphy, G.C.: The structure of features in Java code: An exploratory investigation. In: Ossher, H., Tarr, P., Murphy, G. (eds.) Workshop on Multi-Dimensional Separation of Concerns (OOPSLA 1999) (November 1999)
17. Maccari, A., Heie, A.: Managing infinite variability in mobile terminal software: Research articles. Softw. Pract. Exper. 35(6), 513–537 (2005)
18. Mende, T., Beckwermert, F., Koschke, R., Meier, G.: Supporting the grow-and-prune model in software product lines evolution using clone detection. In: Proc. of the 2008 12th European Conference on Software Maintenance and Reengineering, CSMR 2004, pp. 163–172. IEEE Computer Society, Washington, DC, USA (2008)
19. Parra, C., Cleve, A., Blanc, X., Duchien, L.: Feature-based composition of software architectures. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 230–245. Springer, Heidelberg (2010)
20. Salicki, S., Farcet, N.: Expression and usage of the variability in the software product lines. In: Revised Papers from the 4th Int'l Workshop on Software Product-Family Engineering, PFE 2001, pp. 304–318. Springer, London (2002)
21. Snelting, G.: Reengineering of configurations based on mathematical concept analysis. ACM Trans. Softw. Eng. Methodol. 5(2), 146–189 (1996)
22. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques: Research articles. Softw. Pract. Exper. 35, 705–754 (2005)
23. Thummalapenta, S., Xie, T.: Spotweb: detecting framework hotspots via mining open source repositories on the web. In: Proc. of the 2008 Int'l Working Conference on Mining Software Repositories, MSR 2008, pp. 109–112. ACM, New York (2008)
24. Yang, Y., Peng, X., Zhao, W.: Domain feature model recovery from multiple applications using data access semantics and formal concept analysis. In: WCRE 2009: Proc. of the 2009 16th Working Conference on Reverse Engineering, pp. 215–224. IEEE Computer Society, Washington, DC, USA (2009)