INCORPORATING VARIABILITY DEPENDENCY GRAPHS INTO MULTI-PARADIGM DESIGN WITH FEATURE MODELING

Valentino Vranić
Department of Computer Science and Engineering
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava
vranic@elf.stuba.sk, http://www.dcs.elf.stuba.sk/~vranic

Abstract

Multi-paradigm design enables to map the structures of an application (problem) domain to the appropriate structures (paradigms) supported by a solution domain (programming language). Both application and solution domain, partitioned into subdomains, can be represented as feature models, which is better than the table representation used in the original multi-paradigm design. Feature modeling is not capable of expressing all the important dependencies between the subdomains. Therefore the application of variability dependency graphs, used in the original multi-paradigm design, in addition to feature modeling is proposed here. Since diagrams similar to variability dependency graphs are used in generative programming, as well as feature modeling, multi-paradigm design and generative programming are briefly compared.

1 INTRODUCTION

Recent moves in the field of software development point to the need for multi-paradigm software development in the sense of finding the best way to use the means that are at disposal [5]. The concept of paradigm in software development appears to be important regarding this issue.

The concept of paradigm in the context of software development can be figured at two levels of granularity: large-scale and small-scale [5]. Large-scale paradigms are what is usually considered under the term paradigm, e.g. object-oriented programming. Although seemingly well-defined, after a careful examination, they show up as elusive [3, 5].

The small-scale paradigms appear as more appropriate for the purpose of multi-paradigm software development. They are configurations of commonality and variability [1]. As such, they are akin to the features of programming languages. For example, inheritance is characterized by common behavior and structure, and variability in structure. With respect to the small-scale paradigms, multi-paradigm software development is a metaparadigm; it is a way of deciding which paradigm to use for a given feature of a system or family of systems to be implemented.

Multi-paradigm design for C++ [1] is such a metaparadigm for the paradigms supported by C++. Multi-paradigm design (MPD) can be applied to other solution domains as well; e.g., MPD for AspectJ [4].

MPD for AspectJ is not only an application of MPD to the solution domain of AspectJ; it brings the capabilities of feature modeling into MPD changing it into *MPD with feature modeling*: MPD_{FM}. Section 2 presents MPD_{FM} in a brief detail. Section 3 explains variability dependency graphs and how they can be incorporated into MPD_{FM}. Section 4 recapitulates the article and gives some insight into the relationship between MPD and generative programming.

2 MULTI-PARADIGM DESIGN WITH FEATURE MODELING

A conceptual modeling technique used in domain engineering—known as feature modeling [2]—is akin to commonality and variability analysis used in MPD for C++ [1]. Feature models appear to be more appropriate to capture commonalities and variabilities than the tables that have been originally used in MPD [4].

2.1 Feature Diagrams

Feature diagrams are the key part of a feature model. Besides them, some additional information is provided with each feature [2]. The relevant information for MPD_{FM} that each feature should be accompanied with is: semantic description, rationale, constraints, default dependency rules, binding mode, and instantiation (in this article such information will not be provided explicitly).

A feature diagram, like the one presented in Figure 1, is a directed tree with edge decorations. The root represents a concept, and the rest of the nodes represent features. Edges connect the concept with its features (a feature can be understood as a concept also). There are two types of edges used to distinguish between *mandatory* features, ended by a filled circle, and *optional* features, ended by an empty circle. A concept instance *must* have all the mandatory features and *can* have the optional features.

The edge decorations are drawn as arcs connecting the subsets of the edges originating in the same node. They are used to define a partitioning of the subnodes of the node the edges originate from into *alternative* and *or-features*. A concept instance has exactly one feature from the set of alternative features. It can have any subset or all of the features from the set of or-features.

The nodes connected directly to the concept node are being denoted as its *direct features*; all other features are its *indirect features*, i.e. *subfeatures*. The indirect features can be included in the concept instance only if their parent node is included.

2.2 Transformational Analysis in MPD_{FM}

In MPD_{FM} both domains—application domain, i.e. the domain to which solution techniques is applied (often denoted as problem domain), and solution domain, in which the solution is performed (a programming language)—are represented as feature models. Subsequently, a mapping from application to solution domain, called transformational analysis, can be performed in order to find the appropriate paradigms for application domain features. The results of transformational analysis are then translated into a code skeleton.

Figure 1 depicts an example of transformational analysis of text editing buffer domain (based on an example from [1]). Text editing buffer represent a state of the file being edited in a text editor. It caches the changes until the user saves the text editing buffer into the file. Different text editing buffers employ different working set management schemes and use different character sets. All text editing buffers load and save their contents into files, maintain a record of the number of lines and characters, the cursor position, etc.

The inheritance paradigm of AspectJ is presented in the right-bottom corner of Figure 1. The figure depicts a part of the results achieved in the transformational analysis of the *File* subdomain of text editing buffers. We assume that *File* and its alternative subfeatures that represent the file types, like *database*, *Unix file*, etc., have already been mapped to the class paradigm. While *File* matches with *base type*'s subfeature *Class*, its subfeatures that represent the file types match with *subtype*'s *Class* subfeature. According to this, the relationship between *File* and the file types matches with inheritance.

3 VARIABILITY DEPENDENCY GRAPHS

In MPD, variability dependency graphs are used to show the relationship between domains and their parameters of variation [1]. Despite a very simple notation, they enable the identification of circular dependencies between domains (so-called codependent domains), and the identification of shared domains and their unification (i.e., reduction of variability dependency graphs).

Feature diagrams themselves are not capable of fulfilling these tasks because they are trees and therefore, unlike variability dependency graphs, cannot contain cycles. The edges in feature diagrams do not carry any predefined semantics [2], while the edges of variability dependency graphs have the meaning of "depends on".

If variability dependency graphs are to be derived from feature models, the question is what kind of features corresponds to parameters of variation. Parameters of variation represent the places where the variation in a system family appears: a parameter of variation is an abstraction of the whole range of possibilities among which one can be selected during the creation of a family member.

A category of features that fits into this role is the *singular variation point*. A variation point is a feature to which variable features, i.e. optional, alternative, optional alternative, or or-features, are attached. A singular variation point is such a variation point that allows to include at most one of its direct subfeatures.

The component categories dependency graphs, which are used in generative programming to sort component categories into a GenVoca layered architecture, are similar to variability dependency graphs. These diagrams are also being drawn according to feature models. A node in component categories dependency graphs represents either a component category or configuration repository (a composite node containing all the component categories that all other component categories depend on, but that do not depend on each other), and edges represent "uses" dependency (in the direction of an arrow). A configuration repository can be easily decomposed into a component categories dependency subgraph.

A component category is an abstraction of the component. When generating family members, a concrete component will take place of the component category. According to this, a component category represents a parameter of variation in the sense of MPD, or a singular variation point in the sense of feature modeling. The "uses" dependency has the same meaning as "depends on" relationship in MPD. According to [2], A *uses* B means that B is a *support* domain of A (e.g., "the domain of container packages is a support domain of the domain of matrix packages"), and this means that A *uses* B (i.e., the domain of matrix packages uses the domain of container packages).

4 CONCLUSIONS

This article briefly presented multi-paradigm design with feature modeling (MPD_{FM}). A connection has

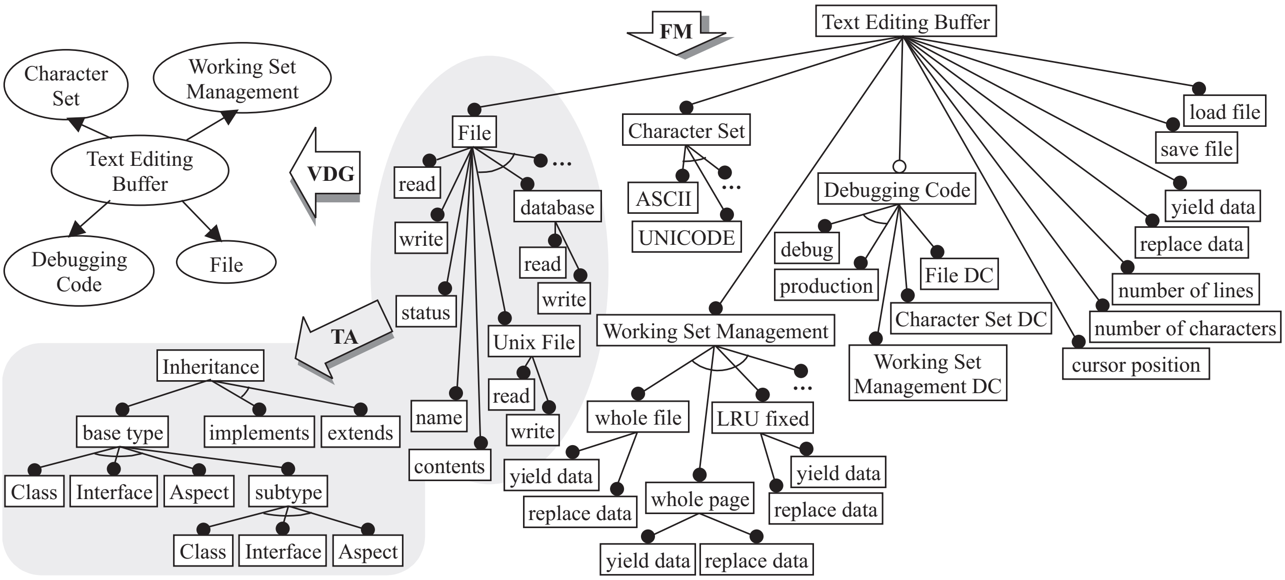

Figure 1: The application domain feature model (FM) is mapped to solution domain in a process of transformational analysis (TA). Variability dependency graphs (VDG) can be obtained from the feature model.

been found between feature models and variability dependency graphs that enables them to be derived directly from the feature diagrams.

Figure 2 shows the main phases of both generative programming and MPD$_{FM}$. The common phases are decorated with gray stripes, the phases appearing only in MPD$_{FM}$ are depicted gray, while the phases appearing only in generative programming are depicted white. The arrows between phases indicate the flow of results. As can be seen from the figure, one could do the domain scoping, feature modeling, and even create dependency graphs without having to decide for either of the two approaches in these early phases.

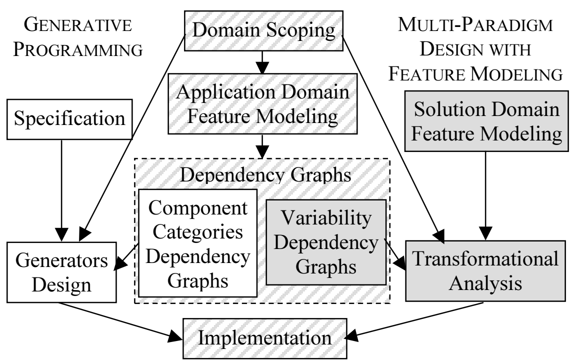

Figure 2: Multi-paradigm design and generative programming.

The real difference between the two approaches becomes more apparent now: while multi-paradigm design helps designer select the appropriate paradigms according to the application domain needs, in generative programming, this selection of paradigms is delegated to the generator.

What has been described in this article is a part of the work on establishing MPD$_{FM}$ as such and MPD$_{FM}$ for AspectJ in particular. Among other issues, the further work on MPD$_{FM}$ embraces incorporating variability dependency graphs into transformational analysis of MPD$_{FM}$ and finding a better way to note the results of transformational analysis.

Acknowledgment. This work was partially supported by Slovak Science Grant Agency, grant No. G1/7611/20.

REFERENCES

[1] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

[2] K. Czarnecki and U. Eisenecker. *Generative Programing: Principles, Techniques, and Tools*. Addison-Wesley, 2000.

[3] V. Vranić. A concept of paradigm in the multi-pradigm software development. In *Proc. of 3rd Scientific Conference on Electrical Engineering and Information Technology for Ph.D. Students (*ELITECH *2000)*, Bratislava, Slovakia, Sept. 2000.

[4] V. Vranić. AspectJ paradigm model: A basis for multi-paradigm design for AspectJ. In J. Bosch, editor, *Proc. of 3rd International Conference on Generative and Component-Based Software Engineering (GCSE 2001)*, LNCS 2186, pages 48–57, Erfurt, Germany, Sept. 2001. Springer.

[5] V. Vranić. Towards multi-pradigm software development. To appear in Journal of Computing and Information Technology (CIT), 2001.