MODELS, FEATURES AND ALGEBRAS
An Exploratory Study of Model Composition and Software Product Lines

Roberto E. Lopez-Herrejon
Computing Laboratory, University of Oxford, U.K.

Keywords: Algebra, feature, Feature Oriented Software Development (FOSD), Model, Model Driven Software Development (MDSD).

Abstract: Software Product Lines (SPL) are families of related programs distinguished by the features they provide. Feature Oriented Software Development (FOSD) is a paradigm that raises features to first-class entities in the definition and modularization of SPL. The relevance of model composition has been addressed in UML 2 with new construct Package Merge. In this paper we show the convergence that exists between FOSD and Package Merge. We believe exploring their synergies could be mutually beneficial. SPL compositional approaches could leverage experience on the composition of non-code artifacts, while model composition could find in SPL new problem domains on which to evaluate and apply their theories, tools and techniques.

1 INTRODUCTION

Software Product Lines (SPL) are families of related programs distinguished by the set of *features*, i.e. increments in program functionality, they provide (Batory et al., 2004). Extensive research has shown how SPL practices can improve factors such as asset reuse, time to market, or product customization and the important economical and competitive advantages they entail (Pohl et al., 2005).

Feature Oriented Software Development (FOSD) is a paradigm that raises features to first-class entities in the definition and modularization of SPL. A fundamental premise of FOSD is step-wise development: constructing complex programs by successively adding features. At the heart of FOSD is a feature algebra that drives the composition of the software artifacts used throughout the software development life cycle. However, to date, this algebra has been applied to several artifact types such as source code and XML files, but has not been thoroughly used for model composition, in particular for UML-based models.

The release of OMG's *Model Driven Architecture (MDA)* initiative has increased the attention to software modeling of software practitioners, researchers, and industry. A promise of MDA is to raise the abstraction level from lower level implementation artifacts to UML models and provide means to transform such models, possibly through several stages, to executable artifacts that can be targeted to different platforms. *Model Driven Software Development (MDSD)* goes a step further by broadening the scope to non-UML based models. A crucial factor to make the Model Driven promise a reality is the development of techniques and tools that formally and precisely abstract, represent and manipulate models and their elements. Of particular importance are mechanisms to break complex models into manageable partial views that can subsequently be composed. Recent work addresses these issues by proposing generic algebraic operations to describe model composition (Herrman et al., 2007). The relevance of model decomposition has also been addressed in UML 2 where new construct Package Merge is used to simplify the specification of the UML metamodel (OMG, 2007).

In this paper we show the significant convergence that exists between research in FOSD and model composition. As an example, we analyze the similarities of Package Merge with the underlying algebra of FOSD. We believe exploring the synergies of both lines of research could be mutually beneficial. Composition approaches for SPL development could leverage experience on the composition of non-code artifacts, while model composition could find in SPL new problem domains on which to evaluate and apply their theories, tools and techniques.

E. Lopez-Herrejon R. (2008).
MODELS, FEATURES AND ALGEBRAS - An Exploratory Study of Model Composition and Software Product Lines.
In *Proceedings of the Third International Conference on Software and Data Technologies - SE/GSDCA/MUSE*, pages 293-296
DOI: 10.5220/0001901302930296

2 FEATURE ORIENTED SOFTWARE DEVELOPMENT

Feature Oriented Software Development (FOSD) is a paradigm that raises features to first-class entities in the definition and modularization of SPL. A fundamental premise of FOSD is step-wise development: constructing complex programs by successively adding features. *AHEAD (Algebraic Hierarchical Equations for Application Design)* is a realization of FOSD (Batory et al., 2004).

AHEAD Tool Suite (ATS) provides support not only for code composition but for several other artifacts such as XML files, grammar files, and equation files. ATS has been used to synthesize large systems (in excess of 250K Java LOC) from program expressions (Batory et al., 2004). In ATS, a feature is implemented as directory that contains all artifact files related to the feature. Artifact types are distinguished by the names of the file extensions and composition is based on file and extension names.

2.1 FOSD Algebra

In FOSD a program consists of a set of composed features, and a Software Product Line is the set of all programs that can be composed from the set of features of the problem domain which satisfy its constraints (Batory et al., 2004). For example let `M = {f, h, i, j}` which means domain `D` has features `f`, `h`, `i`, and `j`. Programs are denoted as follows where • operator corresponds to feature composition:

```
prog1= i•f      // prog1 has features f and i
prog2= j•h      // prog2 has features h and j
prog3= i•j•h    // prog3 has features h,j,i
```

Features are hierarchical modules that can contain any number of nested modules. Two features are composed by recursively composing their nested elements as captured by the following law:

Law of Composition. Let `X` and `Y` be features defined as follows:

$$X = \{a_X, b_X, c_X\}$$
$$Y = \{a_Y, c_Y, d_Y\}$$

Where `a`, `b`, `c`, and `d` are nested features whose subscript indicate the feature they belong to. The composition of `X` and `Y`, denoted as `X•Y`, is:

$$X \bullet Y = \{a_X, b_X, c_X\} \bullet \{a_Y, c_Y, d_Y\}$$
$$= \{a_Y \bullet a_X, b_X, c_Y \bullet c_X, d_Y\}$$

Nested features are composed by names (ignoring the subscripts). The features whose names do not have a match, like b_x or d_y, are simply copied.

```
1  /* Feature Base */
2  layer Base;
3  class Figure { int x,y;  ... }
4
5  layer Base;
6  class Rectangle extends Figure {
7   int width, height;
8   void draw() { ... // base rectangle draw }
9  }
```

```
10 /* Feature Color */
11 layer Color;
12 refines class Rectangle {
13     int color;
14     Pattern background;
15     void draw() {
16       ... // set color and pattern
17       Super().draw();
18     }
19 }
20
21 layer Color;
22 class Pattern { ... }
```

```
23 /* Composition of Feature Base and Color */
24 class Figure { int x,y;  ... }
25 class Rectangle {
26     int color;
27     int width, height;
28     Pattern background;
29     void draw() {
30       ... // set color and pattern
31       ... // base rectangle draw  substitutes
           Super().draw();
32     }
33 }
34 class Pattern { ... }
```

Figure 1: Jak Composition.

2.2 Example

In this section we provide a very simple example to illustrate FOSD feature composition. We will use this example again to illustrate package merge in next section.

Consider a simple product line of graphics applications that consists of two features, `Base` and `Color`, that draw rectangles that can either be empty or filled with a color and a pattern. The ATS uses a language called, *Jak* which is a superset of Java. The implementation of features `Base` and `Color` would look like Figure 1, lines 1-9 and lines 10-22 respectively.

The Jak keyword `refines` indicates that the elements in the class are effectively added to an existing class, `Rectangle` in our case. Also notice the keyword `Super` whose syntax is: `Super(paramtypes).method(actparams);` where `paramtypes` are the formal parameter types, `method` is the method name and `actparams` are the actual parameters. `Super` works in a similar way to standard method overriding but at the feature level. The result of the composition of both features is shown in lines 23-34. In this example, the composition of method `draw` causes the execution of the code that sets the color and pattern (line 30) followed by the base rectangle draw code (line 31).

This composition could also be expressed alge-

braically as follows [1]:

$$
\begin{aligned}
Base &= \{Figure, Rectangle_{Base}\} \\
Figure &= \{x, y\} \\
Rectangle_{Base} &= \{width, height, draw_{Base}\} \\
Color &= \{Rectangle_{Color}, Pattern\} \\
Rectangle_{Color} &= \{color, background, draw_{Color}\} \\
Color \bullet Base &= \{Figure, Pattern, \\
&\quad Rectangle_{Color} \bullet Rectangle_{Base}\} \\
Rectangle_{Color} \bullet Rectangle_{Base} &= \\
&\quad \{width, height, \\
&\quad draw_{Color} \bullet draw_{Base}\}
\end{aligned}
$$

2.3 Algebraic Properties

Feature composition can thus be regarded as the binary function, $\bullet : F \times F \rightarrow F$, where F stands for feature, that exhibits the following properties where `f`, `g`, and `h` are features (Lopez-Herrejon et al., 2006; Apel et al., 2007):

1. *Closure:* The composition of two features is a composite feature.
2. *Associativity:* $(f \bullet g) \bullet h = f \bullet (g \bullet h)$
3. *Identity:* ξ is the representation of an empty feature.
4. *Idempotence:* $f \bullet f = f$
5. *Non-commutativity:* $f \bullet g \neq g \bullet f$
   This is a result of the order of composition when considering method refinements such as method `draw` in our example.

3 PACKAGE MERGE

Package Merge is novel construct of UML 2 which is extensively used to specify the UML 2 metamodel (OMG, 2007). It allows to merge or compose the contents of one package with another and involves three main entities:

- Merged package: The first operand of merge, the packaged to be merged into the receiving package.
- Receiving package: The second operand of merge, conceptually contains the results of the merge.
- Resulting package: The package that conceptually contains the results of the merge.

[1] For simplicity we omit the subscripts when not necessary.

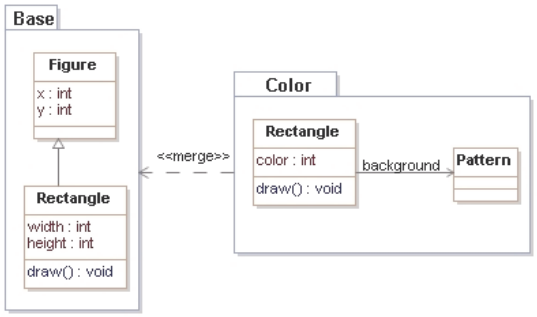

Figure 2: Packages Base and Color.

Notice here the use of the adverb *conceptually* which takes the same connotation in the case of generalization where the subclass conceptually, not textually, contains the elements of the superclass. Also notice here, that the receiving and resulting package are defined to be the same.

The UML specification defines package merge with (OMG, 2007): *match rules* which determine when two elements are to be merged, *constraints* that define the preconditions that must be met for the merge to take place, and *transformations* describe the postconditions of the merge. Among the transformation rules of package merge three of them stand out in our context, transcribed verbatim next:

1. Default rule: Merged or receiving elements for which there is no matching element are deep copied into the resulting package.
2. The result of merging to elements with matching names and metatypes are the exact copies of each other in the receiving element.
3. Matching elements are combined according to the transformation rules specific to their metatype and the results included in the resulting package.

Armed with these specifications we can now illustrate package composition with our example of Section 2.2. Consider now each feature modeled as a package as shown in Figure 2.

Following the rules of package merge, the resulting package would look like Figure 3. Not surprisingly, given that package merge combines elements by name and deep copies elements with no match, the outcome is remarkably equivalent to that of Jak shown in Figure 1.

3.1 Algebraic Properties

Dingel et al. have performed an exhaustive study of package merge (Dingel et al., 2007). Their study found that the UML specification has multiple cases where merge rules are missing or are ambiguously

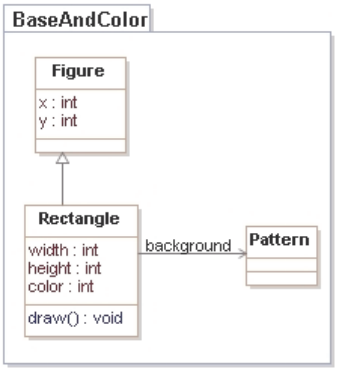

Figure 3: Conceptual Composed Packages.

defined. Their work provides new rules when missing and eliminates existing ambiguities. Furthermore they develop a formal model based in Alloy to check the following properties: uniqueness, associativity, commutativity, and idempotence (Dingel et al., 2007). It must be noted that, from the UML specification, package merge has as identity the empty package and is closed (OMG, 2007).

4 ANALYSIS

We have seen that both feature composition and package merge exhibit remarkably similar algebraic properties. There are however significant differences. In the case of FOSD the order composition is relevant and is made explicit in the order of the operands of feature composition operator •. In the case of package merge there is not an explicit order of composition when a receiving package is merged with multiple merged packages. Another difference is that in package merge the resulting package is always the receiving package, this expressed algebraically using feature composition as: `Receiving = Merged • Receiving` .

5 CONCLUSIONS AND FUTURE WORK

In this paper we show the significant convergence that exists between research in FOSD and model composition, in particular with package merge. We believe exploring the synergies of both lines of research could be mutually beneficial. For instance, FOSD could be extended to compose UML class diagram artifacts which can serve as documentation of the single product applications or be the basis to generate from them other artifacts such as code. As a first step we are implementing a prototype tool that uses Epsilon merging language to perform package merge. Our goal is to integrate it into the ATS.

Much of the tooling effort in MDD today is focused on representing UML-based models and defining model transformations. What is generally lacking are tools to express model composition of UML models by algebraic means. We believe that frameworks such as MOMENT (Boronat et al., 2007)and the work of Romero et. al (Romero et al., 2007) can serve as a foundation on which to implement the FOSD algebra in a framework that encompasses not only models but also code artifacts such as Jak. Building such tools is the subject of our future work.

REFERENCES

Apel, S., Lengauer, C., Batory, D., Moller, B., and Kastner, C. (2007). An algebra for feature-oriented software development. Technical report, University of Passau/MIP-0706.

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371.

Boronat, A., Carsí, J. A., Ramos, I., and Letelier, P. (2007). Formal model merging applied to class diagram integration. *Electr. Notes Theor. Comput. Sci.*, 166:5–26.

Dingel, J., Zito, A., and Diskin, Z. (2007). Understanding and Improving Package Merge. Software and Systems Modeling.

Herrman, C., Krahn, H., Rumpe, B., Schindler, M., and Volkel, S. (2007). An Algebraic View on the Semantics of Model Composition. In *ECMDA-FA*.

Lopez-Herrejon, R., Batory, D., and Lengauer, C. (2006). A Disciplined Approach to Aspect Composition. In *A Disciplined Approach to Aspect Composition. PEPM*.

OMG (2007). Uml infrastructure specification v2.1.2.

Pohl, K., Bockle, G., and van der Linden, F. J. (2005). Software Product Line Engineering: Foundations, Principles and Techniques. In *Springer*.

Romero, J., Rivera, J., Durán, F., and Vallecillo, A. (2007). Formal and Tool Support for Model Driven Engineering with Maude. Journal of Object Technology 6(9):187-207.