

Understanding Feature Modularity in Feature Oriented Programming and its Implications to Aspect Oriented Programming

Roberto E. Lopez-Herrejon
rlopez@cs.utexas.edu
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712

Abstract. My research is in software product lines. My focus is on feature modularity, a basis for product line development, and the cornerstone of *Feature Oriented Programming (FOP)*. Additionally, I have studied the support that different novel modularity technologies provide for features, with emphasis on those categorized as *Aspect Oriented Programming (AOP)*. Also, the relationship between FOP and AOP is a common source of misunderstandings and misconceptions. My dissertation develops an algebraic model that exposes the differences between both programming paradigms and indicates possible venues for mutual benefit.

1 Motivation

A *feature* is a prominent or salient part of an object or thing. Every day objects like cars, houses, or dogs are distinguished among similar objects by the set of features they exhibit such as color, size, or breed. A similar scenario can be applied to programs where features correspond to the functionality that programs provide. For instance, consider a word processor program with typical features of file loading and saving, editing options, spelling checker, font formatting, printing options, and so on.

Abstracting programs in terms of features facilitates their understanding. More importantly, it opens the possibility to think about constructing programs that provide different combinations of features. The set of such programs is called a *software product line*, and the members are thus distinguished by the combinations of features they support.

To carry along with the implementation of product lines, the next step is modularizing features so that we could use them to assemble particular products. Unfortunately, conventional modularization approaches like functions, classes or packages are not appropriate for feature modules. A typical feature implementation spreads over several module (class, package) boundaries. Furthermore, a single module (class, package) may contain intertwined fragments from multiple features.

Programmers must lower their abstractions from features to those provided by the underlying programming languages, a process that is far from simple let alone amenable to significant automatization. Hence the gap between feature abstractions and their modularization severely hinders product line development. The problem is that feature modularity is not well understood and is not well supported in conventional programming languages.

Feature modularity is the cornerstone of *Feature Oriented Programming (FOP)*, a programming paradigm that generalizes and builds on the success of *Relational Query Optimization (RQO)*. The key insight derived from RQO is expressing programs and their designs in terms of relational algebra expressions whose operations are composed and possess mathematical properties. Such program specifications are amenable to *automatic programming* (derivation of optimized programs from unoptimized ones), and *generative programming* (generation of efficient code from higher-level specifications) [10]. FOP raises the level of abstraction from low-level module implementations to mathematical entities that represent features and are subject to algebraic laws. Thus, FOP aims to provide a general algebraic theory of software development in the same sense that relational algebra is a theory upon which query evaluation programs that access relational databases are built.

In recent years, there has been an increasing interest in new modularization techniques to help overcome limitations of conventional Object Oriented Programming. Most of these techniques have been labeled under the *Aspect Oriented Programming (AOP)* paradigm [9]. An *aspect* is a module that implements a *crosscutting*

concern, i.e. its implementation cuts across module boundaries (such as classes) or it is intertwined with fragments of other concerns. The main thrust behind AOP research has been providing ever more expressive mechanisms to describe the crosscutting nature of aspect modules, at the source and bytecode levels, but also at run-time. The result has been a plethora of techniques and tools that appear similar on the surface but that on closer look present differences that could render some approaches infeasible for certain application domains. Thus, understanding the similarities and differences between approaches is important. To achieve such understanding it is necessary to have a model of modularization that aids the comparison. Moreover, how different AOP techniques support building product lines and large scale applications is still an area that have not been thoroughly studied.

The relationship between FOP and AOP has been a source of controversy, misconceptions, and confusion among practitioners of both paradigms and software engineering researchers. Aspect module implementations currently support a wider variety of modularization mechanisms than current feature modules do. Thus it is easy to jump to the conclusion that FOP is a subset of AOP. However, factors other than modularization capability must be considered while evaluating the relationship.

Algebraic models of modules and their composition raise the level of abstraction from implementation details to mathematically supported properties of programs. This fact makes algebraic models suitable mechanisms to compare and contrast modularization and composition approaches from an architectural point of view, and is fundamental for their understanding, comparison, and improvement.

2 Work Completed

2.1 Foundations of FOP

My work helped establish the foundations of FOP, in particular a feature composition model called *Origami* [5]. Origami is an approach to *Multi-Dimensional Separation of Concerns* which regards systems as composition of units implemented in different artifacts that have complex and simultaneous relations in different views or design dimensions [15]. In Origami, programs are defined by combinations of orthogonal specifications that significantly simplify the task of specifying the synthesis of potentially very large and complex programs. The AHEAD tool suite has been implemented by an Origami model [6][7].

2.2 Evaluation of Support for Features in Advanced Modularization Technologies

I performed an evaluation study of modularization and composition of five technologies by comparing the implementations of a simple product line based on the *Expression Problem*, a canonical example in programming languages research. This product line was implemented in AspectJ, Hyper/J, Jiazzi, Scala and AHEAD, all either extensions of Java or Java-like languages¹. The comparison among technologies was made in terms of basic feature definition and composition properties such as the types of program increments they can modularize, how they express composition, etc. Most importantly this study showed that by using an algebraic model apparently dissimilar technologies, at a superficial level, have substantial similarities at the architectural level. For further details see [12].

2.3 An Algebraic Model of Aspects and Their Composition

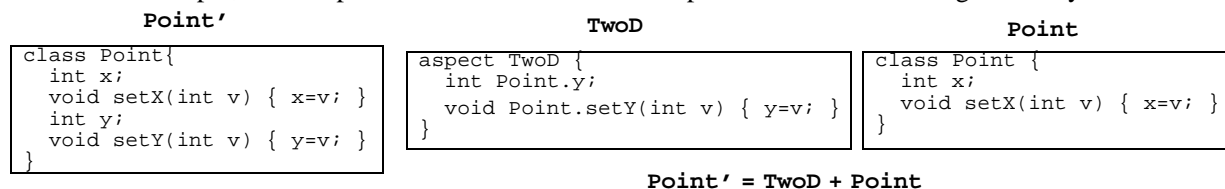
AspectJ [2][11] is the most prominent AOP language; however, it has a complex composition semantics that hinders the use of aspects to build large scale systems in a reusable and incremental fashion. I devised a model that exposes these problems in a clear and concise way. The key insight of this model is to regard aspect crosscuts, both static and dynamic, as *program transformations* that is functions that map programs. Adopting this perspective raises aspects from code artifacts to mathematical entities and enables the development of mathematically-based models of aspects and their composition.

A complete description of this model is outside the scope of this paper. However, the main ideas are sketched below, readers interested in more details should consult [14].

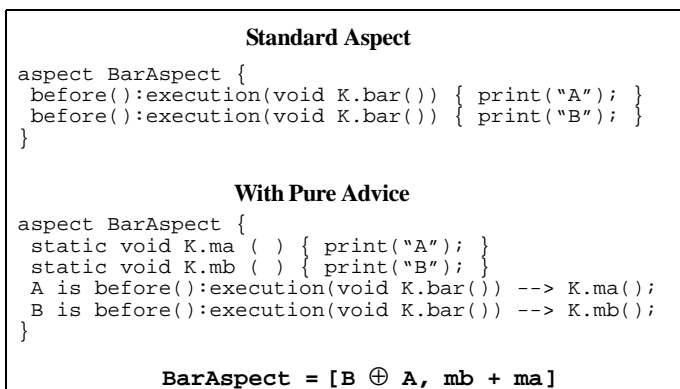
1. MultiJava was considered for the evaluation [8]. However, MultiJava cannot modularize several of the requirements of the features in the case study. For more details see [12].

The model has three operations that build upon the notions of introductions, advice, and weaving.

Introduction Addition. Operation $+$ denotes program addition. It is a binary operation that performs disjoint union on program fragments, which are sets of variables and methods. The introductions of an aspect define a program fragment, as they too form a set of variables and methods. $+$ has an identity, commutes, and is associative. For example, the composition of class `Point` and aspect `TwoD` is denoted algebraically as:



Advice Addition. Operation \oplus denotes advice addition and is used to express advice precedence, the order in which advice is woven. Each advice can be seen as an implicit method declaration with an implicit method call. *Pure advice* makes this distinction explicit. For example, aspect `BarAspect`, shown on the right. It contains two pieces of advice. This aspect can be regarded as two introductions, one for each advice body, and two pieces of pure advice, A and B, that make explicit calls to their corresponding methods. Thus an aspect can be modeled as a vector of two entries, the



first containing a \oplus -summation of pure pieces of advice, and the second containing $+$ -summation of introductions. \oplus has an identity, is not commutative, and is associative. The expression $A \oplus B$ means, first weave advice B and then weave advice A, thus makes advice precedence explicit.

Advice weaving. Operation $*$ denotes advice weaving, applying advice to base code. This operation has an identity, is right-associative, and distributes over advice addition.

Vector Model of AspectJ composition. Operation \diamond denotes AspectJ composition. Let $A_1=[a_1, i_1]$ and $A_2=[a_2, i_2]$ be aspects and $P=[1, p]$ a program. Their composition is expressed as:

$$A_2 \diamond A_1 \diamond P = [a_2, i_2] \diamond [a_1, i_1] \diamond [1, p] = [a_2 \oplus a_1 \oplus 1, i_2 + i_1 + p]$$

The program that results from weaving advice into program fragments of a vector $V=[a, i]$ is expressed as:

$$|V| = |[a, i]| = a * i$$

Thus the program that results from weaving aspect A_1 and A_2 to program P is denoted as follows:

$$|A_2 \diamond A_1 \diamond P| = (a_2 \oplus a_1 \oplus 1) * (i_2 + i_1 + p)$$

In words *all* pieces of pure advice can affect (be woven to) *all* the introductions and programs in the composition. This fact causes problems when using AspectJ for incremental development and limits aspect reuse [13][14].

Functional model of aspect composition. An alternative model that treats aspect composition as function composition. Let $A_1=[a_1, i_1]$ and $A_2=[a_2, i_2]$ be aspects and $P=[1, p]$ a program. Their composition is expressed as:

$$A_2(A_1(P)) = a_2 * (i_2 + a_1 * (i_1 + p))$$

I have shown that the functional model is more expressive than AspectJ's vector model composition model as it can synthesize more programs. A proof is given in [14].

3 Future Work

The underlying motivation of my dissertation is the search of a deeper and more fundamental understanding of program design and structure. I believe that algebraic models are a promising way to achieve this understanding. From my perspective, the goal of software design is capturing and expressing design decisions on a mathematical sound basis which supports automatic generation (generative programming) of efficient code (automatic programming) from an abstract specification of program features.

With this motivation in mind, my dissertation has two goals. The first is to create an algebraic model to understand and evaluate feature modularity in emerging modularization techniques, i.e how they support building product lines. With this model, I compare and contrast FOP and AOP techniques and thus help address their applicability and usefulness in concrete domains.

The second goal is assessing how the proposed algebraic model works in building large scale systems. In order to achieve this goal, it is required to: a) build a prototype tool that implements the model, and b) test it by implementing a large scale system. For the implementation of the prototype tool, I plan to work in collaboration with the *AspectBench Compiler (ABC)* [1][4] research group. ABC has as its source language AspectJ and provides the extensibility and flexibility required for the implementation of our model. For the large case study, I plan to use our tool AHEAD (250K+ LOC). AHEAD code is written in *Jak*, an extension of Java. Therefore, in order to use AHEAD as a case study, it is necessary to convert its code from *Jak* to AspectJ, something that can be achieved with a simple translator. We hope this study will help clarify the relationship between AOP and FOP and shows how can they benefit from each other.

4 References

- [1] Aspect Bench Compiler. www.aspectbench.org
- [2] AspectJ. Programming Guide. aspectj.org/doc/proguide
- [3] AHEAD Tool Suite (ATS). www.cs.utexas.edu/users/schwartz
- [4] Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotak, J., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Building the abc AspectJ compiler with Polyglot and Soot. Technical Report No. abc-2004-4
- [5] Batory, D., Lopez-Herrejon, R.E., Martin, J.P.: Generating Product-Lines of Product-Families. Automated Software Engineering Conference (2002)
- [6] Batory, D., Liu, J., Sarvela, J.N.: Refinements and Multidimensional Separation of Concerns. SIGSOFT (2003)
- [7] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Soft. Engr. June (2004)
- [8] Clifton, C., Leavens, G.T., Millstein, T., Chambers, G.: MultiJava: Modular Open classes and Symmetric Multiple Dispatch for Java. OOPSLA (2000)
- [9] Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley (2004)
- [10] Czarnecki, K., Eisenecker, U.W.: Generative Programming - Methods, Tools, and Applications. Addison-Wesley (2000)
- [11] Laddad, R.: AspectJ in Action. Practical Aspect-Oriented Programming. Manning (2003)
- [12] Lopez-Herrejon, R.E., Batory, D., Cook, W.: Evaluating Support for Feature in Advanced Modularization Techniques. To appear ECOOP (2005)
- [13] Lopez-Herrejon, R.E., Batory, D.: Improving Incremental Development in AspectJ by Bounding Quantification. SPLAT Workshop associated to the AOSD (2005)
- [14] Lopez-Herrejon, R.E., Batory, D.: Tamming Aspect Composition: A Functional Approach. Submitted for publication. Department of Computer Sciences, The University of Texas at Austin, Technical Report TR-05-27
- [15] Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE (1999)