From Crosscutting Concerns to Product Lines: A Function Composition Approach

Roberto E. Lopez-Herrejon
Computing Laboratory
Oxford University
Oxford, England, OX1 3QD
rlopez@comlab.ox.ac.uk

Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
batory@cs.utexas.edu

Abstract

Aspects offer sophisticated mechanisms to modularize crosscutting concerns. Aspect Oriented Programming (AOP) has been successfully applied to many domains; however, its application to product line engineering has not been thoroughly explored. Features are increments in program functionality and are building blocks of software product lines. Work on Feature Oriented Programming (FOP) has shown that a crucial factor to synthesize product lines is composing features by function composition. In this paper we describe a way to emulate function composition using AspectJ for the synthesis of a non-trivial product line, present a general mechanism to support it and highlight its potential reuse benefits. Our study also profiles the role different aspect constructs play in the synthesis of product lines and offers venues of research on the use of aspects in product line implementations.

1 Introduction

Aspects offer sophisticated mechanisms to modularize crosscutting concerns and have been successfully used in many application domains [20]. Product lines are a natural testing ground for *Aspect Oriented Programming (AOP)* as they have unique modularization challenges stemming from commonality and variability management, composability, evolvability, etc. [16]. There has been several studies of AOP for product line implementation and design yet its potential has not been fully explored [2][3][15][17][19][26][27][32] [37].

Features are increments in program functionality and are building blocks of software product lines [18]. *Feature Oriented Programming (FOP)* is a compositional paradigm that raises features to first-class entities in the definition and modularization of product lines. FOP is largely driven by simple mathematics based on function composition, which provides a clean conceptual model for the implementation and formalization of product lines and program synthesis [12][1]. The AHEAD tool suite, an implementation of FOP, has been successfully used to implement non-trivial product lines [10][12].

We are not aware of any product line implemented with aspects that has as many features and LOC as those synthesized by AHEAD (250K+ LOC and 50+ features) [4]. This paper shows how function composition was emulated using AspectJ in the synthesis of a non-trivial product line previously implemented with AHEAD. Furthermore, we present a general mechanism to support function composition and highlight its potential reuse benefits. Our study also profiles the role different aspect constructs play in the synthesis of product lines and offers venues of research on the use of aspects in product line implementations.

2 Quadrilaterals Product Line

We develop a simple example, called the *Quadrilaterals Product Line (QPL)*, to illustrate in the coming sections how product lines can be implemented in AHEAD and AspectJ. This example describes the types of features, composition, and mapping issues we encountered in translating an AHEAD codebase to an AspectJ codebase.

The task at hand is to incrementally build two kinds of quadrilaterals: rectangles and trapezoids. Quadrilaterals have a `draw` operation that displays the lines between the four points that form a quadrilateral. However, rectangles and trapezoids use different lines, styles, and colors. Our example consists of three features.

Base feature. This feature defines: a) class `Quadrilateral` that contains four points and a `draw` operation that draws the lines between the points, b) class `Rectangle` that extends `Quadrilateral`, and c) class `Trapezoid` that extends `Quadrilateral` and overrides method `draw` to add new instructions to make the lines thicker.

Feature `Base` can be implemented as follows:

```
class Quadrilateral {
   Point p1, p2, p3, p4;
   void draw() {... std lines ...}
}
class Rectangle extends Quadrilateral {}
class Trapezoid extends Quadrilateral {
   void draw() {
     super.draw(); ... thick lines ...
   }
}                                                    (1)
```

For sake of simplicity we omit details that differentiate rectangles and trapezoids and focus only on the `draw` methods.

Style Feature. This feature adds a pattern to fill rectangles and replaces operation `draw` in trapezoids to display dashed lines only. The result of composing features `Base` and `Style`

is shown below, where the code that is added by the Style feature is underlined:

```
class Quadrilateral {
   Point p1, p2, p3, p4;
   void draw() {... std lines ...}
}
class Rectangle extends Quadrilateral {
  void draw() { super.draw();… fill pattern … }
}
class Trapezoid extends Quadrilateral {
   void draw() { ... dashed lines ... }
}                                                    (2)
```

A *class refinement* is the modularization of changes made by a feature to a class defined in another feature. Feature Style consists of two class refinements, one for Rectangle and one for Trapezoid. The class refinement of Rectangle overrides method draw defined in Quadrilateral. We refer to this type of refinement as a *method override*. The class refinement of Trapezoid effectively replaces method draw defined for this class in feature Base. We refer to this type of refinement as a *method replacement*.

Color Feature. Color adds a color field and a setColor method to Quadrilateral. It sets the color of rectangles to green and the color or trapezoids to blue.

The result of composing features Base with Style and Color, in that order, yields the following result where the changes caused by Color to (2) are underlined:

```
class Quadrilateral {
   Point p1, p2, p3, p4;
   void draw() {... std lines ...}
   int color;
   void setColor(int c) { color = c; }
}
class Rectangle extends Quadrilateral {
  void draw() { setColor(GREEN);
     super.draw();… fill pattern …
  }
}
class Trapezoid extends Quadrilateral {
   void draw() { setColor(BLUE);
    ... dashed lines ...
   }
}                                                    (3)
```

The Color feature is implemented by three class refinements. The refinement of Quadrilateral adds color and setColor. The refinement of Rectangle extends the functionality of method draw by adding a call to setColor with value GREEN. Similarly, the Trapezoid refinement extends functionality of method draw with a call to setColor with value BLUE. We refer to these last two refinements as a *method extension*. We call the *derived method body* the part of the method extension that it is the result of the composition of all previous class refinements. In (3) this corresponds to the code of method draw that is not underlined.

Product Line Members. Based on the description of our three features, QPL can synthesize the following three products: 1) Base, 2) Base and Style, 3) Base, Style and Color. Though completely reasonable, QPL does not contain program with features Color and Base. Section 4.2 explains an interesting characteristic of this product in the AspectJ implementation of QPL.

Modularization Issues. A straightforward implementation of this simple product line is to use preprocessors, where the code of each feature is surrounded by #if-#endif statements. If a feature is included in a product, its code is included. While the use of preprocessors is common, it is not a first-class language modularization technique. (Preprocessors, by definition, express concepts that are not supported by a host language). A common goal of AHEAD and AspectJ is to define programming language constructs to provide more robust support for crosscut modularity. In the next two sections we evaluate their modularity concepts for implementing QPL. We start on the next section by describing the concepts specific to AHEAD.

3 FOP and AHEAD

FOP aims at developing a structural theory of programs to express program design, manipulation, and synthesis mathematically whereby program properties can be derived from a program's mathematical representation. In this context, a program's design is an expression (i.e. composition of operations from a domain-specific algebra), program manipulation is expression manipulation (such as expression optimization), and program synthesis is expression evaluation. FOP is a generalization of *Relational Query Optimization (RQO)*, arguably the most significant result in automated software engineering [9][12][24]. *AHEAD (Algebraic Hierarchical Equations for Application Design)*, is a realization of FOP that is based on a unification of algebras and step-wise development [1][12].

An AHEAD model of a domain is an algebra that offers a set of operations, where each operation implements a feature. We write M = {f, h, i, j} to mean model M has operations (or features) f, h, i, and j. AHEAD categorizes features as *constants* and *functions*. Constant features represents base programs, those implemented with standard classes and interfaces. For example:

```
f       // a program with feature f
h       // a program with feature h
```

Function features represent *program refinements* or *extensions*, programs that add a feature to the program received as input. For instance:

```
i(x)     // adds feature i to program x
j(x)     // adds feature j to program x
```

The design of a program is a named expression which we refer as a *program equation*. For example:

```
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
```

Thus, the features provided by a program can be determined and properties derived from the expression that defines it. The family of programs that can be created from a model is a product line. In general, not all features are compatible. The use of one feature may preclude the use of some features or may demand the use of others. AHEAD provides mechanisms to expresses and validate such design constraints [8][13].

AHEAD has been used to synthesize large systems (in excess of 250K Java LOC) from program equations [12][10]. A fundamental premise of AHEAD is step-wise development: one begins with a simple program (e.g., constant feature `h`) and builds a more complex program by progressively adding features (e.g., adding features `j` and `i` to `h` in `prog3`) [35]. AHEAD expresses step-wise development mathematically through function composition thus making explicit the order of feature composition as well as the scope to which a program extension applies (i.e., the program that is received as input) [12][11]. AHEAD tools use a language called *Jak* that is a superset of Java to express classes and class refinements.

3.1 QPL Implementation

Base feature. `Base` is a constant feature that has three Jak files (one for each class):

```
layer Base;
class Quadrilateral {
   Point p1, p2, p3, p4;
   void draw() {... std lines ...}
}

layer Base;
class Rectangle extends Quadrilateral {}

layer Base;
class Trapezoid extends Quadrilateral {
   void draw() {
     Super.draw(); ... thick lines ...
   }
}                                                    (4)
```

There are only two differences with the code in `(1)`: the `layer` construct declares that these classes belong to feature `Base`, and Jak's keyword `Super` refers to the superclass of `Trapezoid`.

Style Feature. `Style` is a function feature that modularizes two class refinements (recognized by the keyword `refines`). In the `Rectangle` refinement, modifier `overrides` denotes that method `draw` is overridden and the superclass reference (`super`) in `(2)` is translated as `Super`. In the `Trapezoid` refinement, modifier `new` means that the `draw` method replaces the one defined for that class in feature `Base`.

```
layer Style;
refines class Rectangle {
  overrides void draw() {
      Super.draw(); … fill pattern …
  }
}

layer Style;
refines class Trapezoid {
   new void draw() { … dashed lines … }
}                                                    (5)
```

Color Feature. `Color` is a function feature that modularizes three class refinements. Class refinement `Quadrilateral` adds new members `color` and `setColor()`. Class refinements `Rectangle` and `Trapezoid` extend their `draw` methods by setting their corresponding colors and calling the `draw` method using keyword `Super`, underlined in the code below. One way to think about `Super` in method extensions is as a placeholder for the derived method body mentioned in `(3)`.

```
layer Color;
refines class Quadrilateral {
   int color;
   void setColor(int c) { color = c; }
}

layer Color;
refines class Rectangle {
   void draw(){ setColor(GREEN); Super.draw();}
}

layer Color;
refines class Trapezoid {
   void draw() { setColor(BLUE); Super.draw();}
}                                                    (6)
```

Product Line Members. One of the tools of AHEAD is a feature composer that receives as input an equation that describes a set of the features and the order in which to compose them. For instance the product member with features `Base`, `Style`, and `Color` (that we can call `All`) is specified as:

```
composer -target=All Base Style Color
```

This composer can also verify that composition is valid according to design rules. For further details consult [1]. In Section 6 we elaborate on the advantages of function composition in product line designs.

4 AspectJ

AspectJ [6] is an extension of Java whose goal is to modularize *aspects*, concerns that crosscut traditional module boundaries such as classes and interfaces that would otherwise be scattered and tangled with the implementation of

other concerns [7]. AspectJ has two types of crosscuts: static and dynamic.

Static crosscuts affect the static structure of a program [7][22]. Examples are *introductions*, also known as *inter-type declarations*, that add fields, methods, and constructors to existing classes and interfaces.

Dynamic crosscuts run additional code when certain events occur during program execution. The semantics of dynamic crosscuts are commonly understood and defined in terms of an event-based model [23][36]. As a program executes, different events fire. These events are called *join points*. Examples of join points are: variable reference, variable assignment, execution of a method body, method call, etc. A *pointcut* is a predicate that selects a set of join points. *Advice* is code executed before, after, or around each join point matched by a pointcut.

4.1 QPL Implementation

We now show how QPL is implemented by aspects. The techniques we use are identical to those needed to translate an AHEAD code base to an AspectJ code base in Section 5.

Base Feature. This feature is implemented as `(1)`. In AspectJ, standard Java classes and interfaces are referred to as *base code*. AspectJ does not provide a mechanism to modularize multiple classes or aspect files into features [27]. In our implementation, we use file directories to group the classes and aspects that constitute features.

Style Feature. This feature is implemented with two aspects. The first one uses an introduction to override method `draw` inherited from `Quadrilateral`. The aspect is declared `privileged` to have access to private members of the classes involved and the name is formed with the feature and class names implemented by the aspect. The rationale behind these two decisions is explained shortly.

```
privileged aspect Style_Rectangle {
  void Rectangle.draw() {
    super.draw(); … fill pattern …
  }
}                                                    (7)
```

The second aspect uses an `around` advice on the `execution` of method `draw` on `Trapezoid`. For simplicity, we introduced a new method `style$draw` to `Trapezoid` to hold the body of the method replacement. Thus, any time `draw` is executed, the call is intercepted and redirected to method `style$draw`, effectively replacing the original `draw` method for this class in `Base` feature. The pointcut descriptor of the advice contains a `target` clause to obtain a reference, we call it `obj$Trapezoid`, of the object whose method is executed. This reference is used to access instance members on that object thus the need for `privileged` aspects in case members are private. In our example, we use this reference to call method `style$draw`.

```
privileged aspect Style_Trapezoid {
 void around(Trapezoid obj$Trapezoid) :
    execution(void Trapezoid.draw()) &&
     target(obj$Trapezoid){
       obj$Trapezoid.style$draw();
 }
 void Trapezoid.style$draw(){… dashed lines … }
}                                                    (8)
```

Arguably, this feature could be implemented in a single aspect that combines the contents of aspects `(7)` and `(8)`. In general, this approach can cause problems because of precedence rules. We explain why shortly.

Color Feature. This feature is implemented with three aspects. The first one introduces `color` and `setColor` to `Quadrilateral`.

```
privileged aspect Color_Quadrilateral {
  int Quadrilateral.color;
  void Quadrilateral.setColor(int c) {color=c;}
}                                                    (9)
```

The second aspect uses `around` advice on `execution` of method `draw`. It also uses a `target` clause to get the object being called and uses this reference to set the color to `GREEN`. As opposed to the advice in feature `Style`, this aspect uses AspectJ `proceed` statement to continue with the execution of the method. Thus, any time that `draw` is called, the call is intercepted, the color is set, and the execution is resumed.

```
privileged aspect Color_Rectangle {
 void around(Rectangle obj$Rectangle) :
    execution(void Rectangle.draw()) &&
     target(obj$Rectangle){
      obj$Rectangle.setColor(GREEN);
      proceed(obj$Rectangle);
 }                                                   (10)
```

One way to think about `proceed` in method extensions is as a placeholder of the derived method body we mentioned in `(3)` because it indicates the execution of the rest of extensions made to this method by other features.

The third aspect follows along the same lines of the previous one only applied to `Trapezoid` instead of `Rectangle`.

```
privileged aspect Color_Trapezoid {
 void around(Trapezoid obj$Trapezoid) :
    execution(void Trapezoid.draw()) &&
     target(obj$Trapezoid){
      obj$Trapezoid.setColor(BLUE);
      proceed(obj$Trapezoid);
 }                                                   (11)
```

Product Line Members. AspectJ compiler (weaver) `ajc`, uses the file names of base code and aspects of the features to create product members. *Aspect precedence* determines the order aspect introductions and pieces of advice are woven to base code.

In the case of introductions, the one with higher precedence overrides (replaces) those with lower precedence. An introduction cannot override a member already present in base code. This is why class refinement `Trapezoid` in feature `Style` is implemented with an `around` advice.

In the case of pieces of advice, when several of them apply to the same join point, AspectJ follows ordering rules that depend on where the conflicting pieces of advice are defined [7][22]. However, these rules can lead to undefined orders (programmers cannot easily infer the order by looking at the code), circularity errors (compiler cannot not infer a weaving order), and some composition orders cannot be expressed [29].

To prevent problems with advice weaving order, we use only `around` advice and a define an aspect with a `declare precedence` clause to order pieces of advice by features. For instance, the following aspect specifies the weaving order required to obtain the composition of `Base`, `Style`, and `Color` in `(3)`:

```
aspect Ordering {
  declare precedence: Color_*, Style_*;
}                                              (12)
```

This aspect instructs the compiler to weave first the aspects from the `Style` feature and then those from the `Color` feature, according to the ordering rules that apply to `around` advice in different aspects [7][22]. This is the reason why we followed the convention of including feature names as part of the names of our aspects.

Finally, the `ajc` command for the composition in `(3)` is (we omit feature directory names for simplicity)[1]:

```
ajc Quadrilateral.java Rectangle.java
  Trapezoid.java
  Style_Rectangle.java Style_Trapezoid.java
  Color_Quadrilateral.java
  Color_Rectangle.java Color_Trapezoid.java
  Ordering.java                                (13)
```

The order of files in this command is immaterial. Composition validation according to design rules is not part of `ajc`.

4.2 Choice of Pointcuts

AspectJ provides a vast array of pointcuts that could be used for feature implementation. We use `execution` pointcuts because their semantics most closely resembles what AHEAD uses for method extension and replacement in QPL. However, there may be cases where other pointcut combinations could be more appropriate.

For example, consider QPL product that contains `Base` and `Color` features. From a design point of view, this program seems completely reasonable and valid; however, its implementation is not possible using `execution` pointcuts. This is because AspectJ semantics specifies that when a declaring type is used in an `execution` pointcut this captures only join points that match methods declared or overridden in the declaring type [6][7]. In our example, this means that pointcut `execution(void Rectangle.draw())` of feature `Color` in `(10)` matches join points only if `Rectangle` declares or overrides method `draw`, which is not the case in feature `Base` as `Rectangle` inherits the method from `Quadrilateral (1)`. This problem can be solved replacing the `execution` pointcut by a combination of `call` and `target` pointcuts.

This example suggests the existence of patterns of method extensions and replacements that could be better expressed with different combinations of pointcuts, new pointcut constructs tailored for specific feature extensions, or semantically modified versions of existing pointcuts [14]. We believe this is an interesting venue for future research.

4.3 Emulating Function Composition

Function composition makes explicit the order of composition and binds the scope to which program extensions are applied. Consider for instance, the program described in Section 4.1. It can be denoted as function composition `Color(Style(Base))` because:

- Composition order: Aspect `Ordering` in `(12)` tells the weaver to first apply to `Base` the pieces of advice and introductions in `Style` followed by those of `Color`.
- Bounded scope: Feature `Style` applies its extensions only to `Base`, and feature `Color` applies its extensions only to its input program `Style(Base)`.

Elaborating more on the second bullet, feature `Style` overrides method `draw` in class `Rectangle (7)`, and replaces method `draw` in class `Trapezoid` using advice `(8)`. In the first case, the introduction overrides an inherited method in `Rectangle` of feature `Base`. In the second case, the advice captures join points triggered by the execution of method `draw` of `Trapezoid` also in feature `Base`. Thus feature `Style` applies its extensions only to elements in feature `Base`.

Similarly, feature `Color` applies its extensions only to the program it receives as input `Style(Base)`. Its three refinements (`(9)`,`(10)`, and `(11)`) add and extend elements present in feature `Base` and thus are present in `Style(Base)`. Figure 1 illustrates this program. The scoping arrows depict the code that class refinements affect. For example, the two refinements in feature `Style`, affect their corresponding base code classes in feature `Base`. Note that all scoping arrows point upwards, which means that features apply their changes to the programs they receive as input.

1. It should be `Base/Quadrilateral.java`, `Base/Rectangle.java`, `Base/Trapezoid.java`, etc.

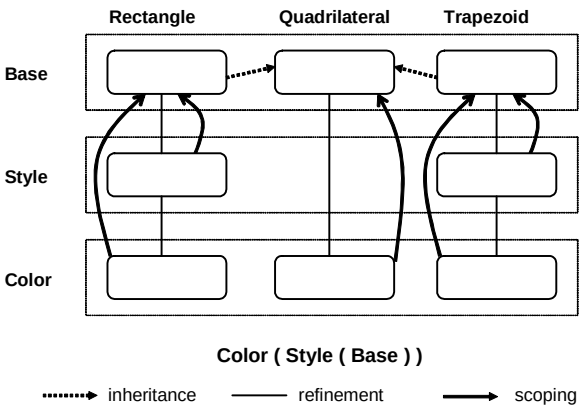


Figure 1. Function Composition in QPL

For the implementation of QPL, precedence and simple pointcuts (capture join points of `Base`) were enough to emulate function composition. However, these two conditions are not enough in general. In Section 6 we discuss why is that the case, illustrate the benefits of function composition of aspects and present a way to achieve it.

5 AHEAD Case Study

The last two sections suggest a mapping between AHEAD and AspectJ constructs, thus opening the possibility of implementing in AspectJ product lines previously built with AHEAD. To the best of our knowledge, we are not aware of any product line in AspectJ of scale comparable to those generated with AHEAD [4]. Thus, the driving goal of our case study is to assess and compare how AspectJ tackles product lines of such scale.

One of the largest product lines synthesized with AHEAD is the AHEAD tool suite itself. It consists of several stand alone and language-extensible tools [1]. We translated into aspects the code base of the five key tools of AHEAD: a) *mixin* performs mixin composition on features, b) *jampack* composes features by collapsing their refinement chains, c) *unmixin* propagates changes from composed files back to their constituent features, d) *jak2java* translates Jak programs into Java, and e) *mmatrix* supports AHEAD feature browser.

5.1 Mapping Jak to AspectJ

QPL illustrates the kinds of features found in our case study. It also indicates a straightforward mapping between Jak and AspectJ constructs that we implemented in a translator called *jak2aj*. This mapping is summarized in Figure 2. Though not shown in the figure, the translation of interfaces is similar to that of classes.

There are four special cases in this mapping:

1. Translation of static methods does not have `target` clauses because their execution is associated to a class not to an object.
2. Methods with arguments `and` (`&&`) an extra `args` pointcut for the method parameters which are bound to the around advice parameters.
3. AspectJ does not permit introduction of protected members [6]. We translated those as public members.
4. Interfaces in `implements` clauses of class refinements are mapped to `declare parent` clauses in their corresponding aspect.

AspectJ has an asymmetrical approach to overriding [6][7]; precedence can override introductions but not base code members. Because of this, we had to distinguish between method override and method replacement.

	Jak	AspectJ
Standard Class	layer L; class C { … }	class C { …}
New field	layer L; refines class C { mods type f; }	privileged aspect L_C { mods type C.f; }
New method	layer L; refines class C { mods type f(args) {...} }	privileged aspect L_C { mods type C.f(args) {...} }

Method override	layer L;
	refines class C {
	overrides mods type f(args) { … Super.f(args); … }
	}
		privileged aspect L_C {
		mods type C.f(args) { … super.f(args); … }
		}

Method replacement	layer L;
	refines class C { new mods type f() { … } }
		privileged aspect L_C {
		type around(C obj$C) : execution(mods type C.f())
		&& target(obj$C) { return obj$C.L$f( ); }
		type C.L$f() { … }
		}

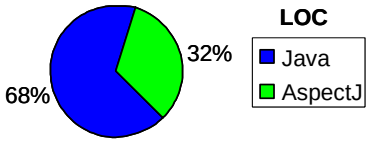
Method extension	layer L;
	refines class C { mods type f() { …Super.f(); … } }
		privileged aspect L_C {
		type around(C obj$C) : execution(mods type C.f())
		&& target(obj$C) { …proceed(obj$C); … }
		}

Figure 2. jak2aj Translation Summary

Tool	Features	LOC
mixin	16	40402
jampack	20	44234
unmixin	11	38798
jak2java	17	43907
mmatrix	12	39712
	Total	207053

(a)

	Java	AspectJ
Num Files	524	503
LOC	38300	18427



(b)

	Java	AspectJ – LOC
Field	1006	58 – 58
Const	40	0 – 0
Method	2238	610 – 9295
Override		164 – 1574
Replace		8 – 440
Extension		8 – 119

(c)

Figure 3. AHEAD Case Study Summary

5.2 Results

The five tools studied are built from combinations of 48 different features. The code generated for all of them is slightly more than 205K+ LOC (Figure 2a). The 48 features surveyed are implemented in 524 standard Java files (base code) and 503 aspect files. In terms of LOC, Java code is 38K+ and AspectJ code 18K+. Thus, we found a 68%-32% ratio between Java code and AspectJ code as illustrated in Figure 2b. The Java code has 1006 fields, 40 constructors and 2200+ methods. AspectJ code introduces 58 new fields and 610 new methods, with 164 method overrides, 8 method replacements, and 8 method extensions (Figure 2c). These numbers and their corresponding LOC indicate that AHEAD tool suite relies heavily on introductions, and only uses a tiny number of pieces of advice. Of the 56700+ LOC in the 48 features of the synthesized tools, only 119 lines (.2% — 2 tenths of one percent) is due to method extension, and 440 lines (.7% — seven tenths of one percent) is due to method overriding.

A strength of AOP is without doubt the sophisticated mechanisms to specify complex pointcuts yet in the case of AHEAD tool suite they were not fully exploited. This is not surprising, as AHEAD itself offers very limited advising capabilities (method and constructor extensions). This explains why less than 1% of the synthesized code is due to advice. An open research question is: if AHEAD had more powerful forms of pointcuts and advice, how much larger would this percentage be?

A bound on this number is the recognition that features in product lines generally implement collaborations [5][34], which in the AOP literature corresponds to *heterogeneous crosscuts*. Such crosscuts deal with field and method introductions and advice whose pointcuts qualify a single join point. AspectJ excels in realizing *homogenous crosscuts* (advice whose pointcuts qualify multiple join points). The important role of heterogeneous crosscuts is not surprising: large programs are not synthesized by adding the same piece of code in different places, but rather, adding different pieces of code in different places. Never-the-less, an open question remains: How can AOP mechanisms be harnessed more fully in the implementation of product lines? We address this question in next section.

6 Function Composition in AspectJ

We have seen that in the context of AspectJ programs function composition implies:

- Making explicit the composition order of base code, introductions and advice at the feature level.
- Scoping features to apply their introductions and advice only to the programs they receive as input.

At first glance these two conditions seem to restrict rather than to benefit AspectJ program development. Our previous work has shown that not to be the case [29]. We illustrate why next.

6.1 Extending QPL

Let us assume that feature `Style`, `(7)` and `(8)`, also contains font type information implemented with a field introduction and a method introduction to class `Quadrilateral`:

```
privileged aspect Style_Quadrilateral {
  int Quadrilateral.font;
  void Quadrilateral.setFont(int f){ font=f;}
}                                                    (14)
```

Furthermore, let us add a new feature to QPL that performs one of the quintessential AOP operations, `Logging`. In our example we want to log the execution of set methods. A straightforward way to implement this feature is as follows:

```
privileged aspect Logging {
void around():
   execution(* void Quadrilateral.set*()) {
     Log(MESSAGE);
     proceed();
   }
}                                                    (15)
```

Consider what happens when we compose these two extensions of QPL, `(14)` and `(15)`, with the original one `(13)`. The resulting `Quadrilateral` class with the woven advice (underlined) is:

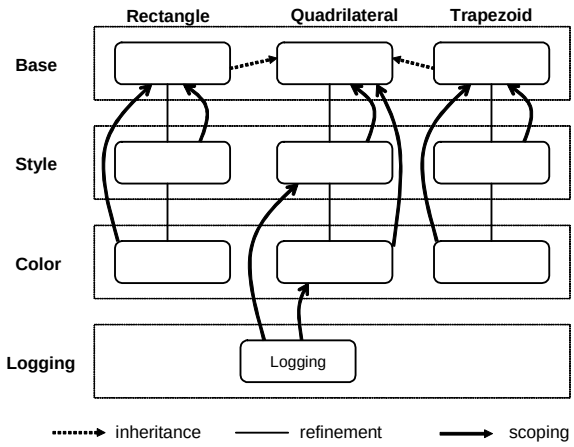


Figure 4. Unbounded Quantification

```
class Quadrilateral {
  Point p1, p2, p3, p4;
  void draw() {... std lines ...}
  int font;
  void setFont(int f){ Log(MESSAGE); font=f;}
  int color;
  void setColor(int c){ Log(MESSAGE); color=c;}
}                                              (16)
```

The logging advice is woven to both set methods, even if the `declare precedence` clause in `Ordering (12)` is modified to include aspect `Logging`. Thus, the scope of advice in `Logging` is global to the composition. We call this phenomenon *unbounded quantification* [28][29], and it stems from the adherence of AspectJ to global reasoning [21]. A consequence of unbounded quantification is that AspectJ can only express with these features (without modifications) the composition illustrated in Figure 4, where `Logging` is at the bottom because it affects `Quadrilateral` refinements in `Style` and `Color` that both add `set` methods.

This example shows that precedence clauses do not provide a general mechanism to enforce function composition. Most importantly, it raises the questions: How can aspects such as `Logging` be composed in an order different from the default? Are there any benefits w.r.t product lines in doing that?

6.2 Bounded Quantification and Feature Reuse

Function composition in AspectJ can be achieved by program equations and *bounded quantification* [28][29]. The first specifies a composition order[2]. The second scopes pieces of advice such that they capture only join points that result from the execution of the program they receive as input according to a program equation.

2. The algebraic model and equation details are outside the scope of this paper, interested readers are encouraged to consult [29].

Suppose we want to compose `Logging` between `Style` and `Color (15)` as in the following program equation:

```
Color ( Logging ( Style ( Base ) ) )
```

Figure 5 illustrates this composition. There are a couple of differences compared to Figure 4. Obviously, `Logging` does not appear at the bottom of the figure. But most importantly, the scoping arrow of `Logging` points only to the class refinement of `Rectangle`, in other words method `setColor` of `Quadrilateral` is outside its quantification scope. Notice also that the rest of the scoping arrows point upwards. What this means is that the scope of the advice and introductions present in the features is bounded to the programs they receive as inputs.

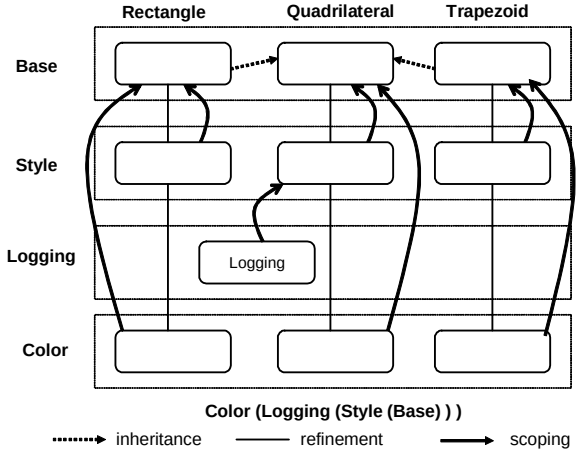


Figure 5. Bounded Quantification

The benefit of function composition is that it allows to build another member of QPL (the program that has the four features and logs only the execution of method `setFont`) that AspectJ composition model *cannot* build without changing the code of `Logging`. Why? Because AspectJ always applies `Logging` globally (to all base code and aspect code of a composition) so the only program synthesizable without intrusive changes is the one depicted in Figure 4. Put it in another way, if we want to build our new program that logs `setFont` executions we need another version of `Logging`:

```
privileged aspect Logging {
void around():
   execution(* void Quadrilateral.setFont()) {
     Log(MESSAGE);
     proceed();
   }
}                                              (17)
```

Unbounded quantification is a special case of bounded quantification. Bounded quantification enables programmers to make finer distinctions in program design which is important in product line development. Bounded quantification also promotes aspect reuse. Even though the above QPL example is small — the programs in Figure 4 and

Figure 5 can be produced by composing QPL features ***as is*** in different orders; AspectJ requires an aspect to be modified to produce both programs, thus showing that ***as is*** feature reuse is more difficult when unbounded quantification is used. We conjecture that our finding will become more relevant when extrapolating to larger product lines with larger sets of features that rely on advice with more elaborate pointcuts, where fine design distinctions and feature reuse are important. This is a research venue we are currently exploring.

7 Related Work

The use of AOP as an implementation technology for product lines have been analyzed and evaluated in several case studies: embedded systems [15][26], graph algorithms [27], extensibility problem [27], mobile phones applications [3][2], middleware software [17][37], and e-commerce [32]. We elaborate on those closest to our work.

Anastasopoulus and Muthig propose criteria to evaluate AOP as a product line implementation technology [3]. Their criteria encompasses the main activities of both framework and application engineering. They regard AOP as a program transformation technique. From that perspective, their evaluation on *Reuse Over Time* concludes that aspect reuse is hindered because it is hard to predict the effects of AOP transformations. The foundations of our function composition model is an algebra that also regards aspects as transformations [29]. This perspective allowed us to analyze AspectJ composition and propose an alternative model with simpler and more predictable composition semantics. Another evaluation criterion is *Variation Type* where authors mention precedence as a mechanism to address variation order. In this paper we showed that in some cases precedence clauses are not enough to express some variation orders.

Alves et al. propose a methodology that combines reactive and extractive approaches to product line development [2]. It is an iterative process that starts by identifying concerns in a set of related products, builds a concern graph [33], and extracts the commonality and variability present in the graph to build a product line design. At each iteration the product line is reactively adapted through code transformations based on a set of template-oriented AOP refactorings. This contrasts with our work as we use aspects as building blocks of an existing product line. However, we believe that a function composition model can also increase reuse in the aspects extracted following their methodology.

Coyler and Clement implemented members a middleware product line using aspects [17]. They developed three features that are typical AOP applications: tracing, monitoring and failure data capture. They also refactored into a feature the support for EJB in a server application. Their implementation of this latter feature shows a 3-1 ratio between the number of introductions and pieces of advices. More surprisingly, the pieces of advice of this feature capture only a single join point as we did in our case study.

A similar study was performed by Zhang and Jacobsen [37]. They refactored aspects from a CORBA implementation using an iterative process they called *Horizontal Decomposition (HD)*. They achieved a 40% reduction on code size and good performance improvement. Liu and Batory have proposed an algebraic theory of feature composition and decomposition that generalizes and provides a mathematical foundation in which to understand HD [25]. This theory also relies on function composition.

There are several approaches to improve variability and feature reuse. *Framed aspects* merge frame technology and AOP [30]. AOP is used to modularize crosscutting concerns while frame technology is utilized for configuration, validation and variability via parameterization, conditional compilation and code generation. Mezini and Ostermann developed *CaesarJ* to improve variability in FOP and AOP [31]. CaesarJ provides *Aspect Collaboration Interfaces (ACI)*, interface definitions for aspects whose purpose is to separate an aspect implementation from its binding. In this way, many aspects can implement the same interface. Both approaches are built upon the unbounded quantification model of AspectJ.

Aspectual Mixin Layers (AML) capitalizes on the strengths of FOP and AOP [5]. In this context, features are mixin layers that contain mixin classes and aspects that can also refine pointcuts and pieces of advice. AML follows a function composition based on bounded quantification. A prototype tool is under development.

8 Conclusions and Future Work

Using aspects to build product lines is an interesting research topic still in its infancy. We translated an AHEAD code base of a tool product line (i.e., the tools that implement the AHEAD tool suite) into an AspectJ code base. To our knowledge, this is one of the largest case studies in product lines and program synthesis to use aspects.

A crucial concept in synthesizing programs in AHEAD product lines is composing features by function composition. We emulated function composition in AspectJ by a combination of a disciplined use of precedence and a careful selection of a small subset of advice. Even so, these two conditions are insufficient to support function composition in a general case. We proposed bounded quantification and algebraic specification (program equations) as a model more suited for product lines and aspects, and described the potential benefits of this approach for feature reuse in product line development.

An interesting statistic from our study was an enormous emphasis on introductions and a very small use of advice (i.e., less than 1 percent) in our tool code base. While the explanation is obvious — the AHEAD tools provide and use only a limited form of advice — it would be very interesting to know how much larger this percentage would be if more sophisticated advice were available, in order to more fully understand the importance of aspects in product lines. We and others [5] conjecture that the predominant use of aspects in product lines will be to implement heterogeneous crosscuts — collaborations whose implementations rely on introductions and advice whose pointcuts qualify single join points — rather than homogenous crosscuts (advice whose pointcuts qualify many join points). Further work is needed to verify this conjecture.

Acknowledgements. We thank Oege de Moor for his help in clarifying subtle issues of pointcut semantics. This research is sponsored in part by NSF's Science of Design Project #CCF-0438786.

9 References

[1] AHEAD Tool Suite (ATS).*www.cs.utexas.edu/users/schwartz*

[2] V. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho, "Extracting and Evolving Game Product Lines", *SPLC* 2005.

[3] M. Anastasopoulus, and D. Muthig, "An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology", *ICSR* 2004.

[4] AOSD Europe Network of Excellence. *http://www.aosd-europe.net*

[5] S. Apel, T. Leich, and G. Saake, "Aspectual Mixin Layers: Aspects and Features in Concert", *ICSE* 2006.

[6] AspectJ, version 1.2.1, *http://eclipse.org/aspectj/*.

[7] AspectJ Manual, *http://www.eclipse.org/aspectj/doc/prog-guide/language.html*.

[8] D. Batory, and B.J. Geraci. "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, Feb. 1997.

[9] D. Batory, G. Chen, E. Robertson, T. Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE TSE*, May 2000.

[10] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder, "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study", *ACM TOSEM*, April 2002.

[11] D. Batory, R.E. Lopez-Herrejon, and J.P. Martin, "Generating Product-Lines of Product-Families", *ASE* 2002.

[12] D. Batory, J.N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement", *IEEE TSE*, June 2004.

[13] D. Batory, "Feature Models, Grammars, and Propositional Formulas", *SPLC* 2005.

[14] O. Barzilay, S. Tyszberowicz, Y. A. Feldman, and A. Yehudai," Call and Execution Semantics in AspectJ", *FOAL Workshop AOSD*, 2004.

[15] D. Beuche, and O. Spinczyk, "Aspect-Oriented Product Line Development in Constrained Environments", *Workshop on Reuse in Constrained Environments OOPSLA* 2003.

[16] P. Clements, and L. Northrop, *Software product lines : practices and patterns*, Addison-Wesley, 2002.

[17] A. Coyler, and A. Clement, "Large-scale AOSD for Middleare", *AOSD* 2004.

[18] K. Czarnecki, and U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[19] Early Aspects website. *http://www.early-aspects.net/*

[20] R.E. Filman, T. Elrad, S. Clarke, M. Aksit, *Aspect-Oriented Software Development*., Addison-Wesley, 2004.

[21] G. Kiczales, and M. Mezini. "Aspect-Oriented Programming and Modular Reasoning", *ICSE* 2005.

[22] R. Laddad, *AspectJ in Action. Practical Aspect-Oriented Programming*, Manning, 2003.

[23] R. Lämmel, "Declarative Aspect-Oriented Programming", *PEPM* 1999.

[24] J. Liu, and D. Batory, "Automatic Remodularization and Optimized Synthesis of Product-Families", *GPCE* 2004.

[25] J. Liu, and D. Batory, "Feature Oriented Refactoring of Legacy Applications", *ICSE* 2006.

[26] D. Lohmann, and O. Spinczyk, and W. Schröder-Preikschat, "On the Configuration of Non-Functional properties in Operating System Product Lines", *ACP4IS Workshop AOSD* 2005.

[27] R.E. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating Support for Features in Advanced Modularization Techniques", *ECOOP* 2005.

[28] R.E. Lopez-Herrejon, and D. Batory. "Improving Incremental Development in AspectJ by Bounding Quantification", *SPLAT Workshop AOSD*, March 2005.

[29] R. E. Lopez-Herrejon, D. Batory, and C. Lengauer, "A disciplined approach to aspect composition", *PEPM* 2006.

[30] N. Loughran, and A. Rashid,"Framed Aspects: Supporting Variability and Configurability for AOP", *ICSR* 2004.

[31] M. Mezini, and K. Ostermann, "Variability Management with Feature-Oriented Programming and Aspects", *FSE-12 ACM SIGSOFT*, 2004.

[32] A. Nyßen, S. Tyszberowicz, and T. Weiler, "Are Aspects Useful for Managing Variability in Software Product Lines? A Case Study", *Aspects and Product Lines Workshop SPLC*, 2005.

[33] M. Robillard, and G. Murphy, "Concern graphs: Finding and describing concerns using structural program dependencies", *ICSE* 2002.

[34] Y. Smaragdakis, and D. Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs", *ACM TOSEM*, April 2002.

[35] N. Wirth, "Program Development by Stepwise Refinement", *CACM* 14 #4, 221-227, 1971.

[36] M. Wand, G. Kiczales, and C. Dutchyn, "A Semantics for Advice and Dynamic Join Points in Aspect Oriented Programming", *TOPLAS* 2004.

[37] C. Zhang, and H. Jacobsen, "Resolving Feature Convolution in Middleware Systems", *OOPSLA* 2004.