Accepted for Publication in ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)

A Disciplined Approach to Aspect Composition

Roberto Lopez-Herrejon, Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{rlopez, batory}@cs.utexas.edu

Christian Lengauer
Fakultät für Mathematik und Informatik
Universität Passau
Passau, Germany
lengauer@fmi.uni-passau.de

Abstract

Aspect-oriented programming is a promising paradigm that challenges traditional notions of program modularity. Despite its increasing acceptance, aspects have been documented to suffer limited reuse, hard to predict behavior, and difficult modular reasoning. We develop an algebraic model that relates aspects to program transformations and uncovers aspect composition as a significant source of the problems mentioned. We propose an alternative model of composition that eliminates these problems, preserves the power of aspects, and lays an algebraic foundation on which to build and understand AOP tools.

1 Introduction

Aspect-oriented programming (AOP) is a promising paradigm that challenges and enhances traditional notions of program modularity [20]. It has been widely applied to different languages but the most influential implementation is AspectJ [7][20][27]. AspectJ has sophisticated and powerful modularization mechanisms that bring clear benefits over traditional modules but also has equally significant drawbacks. Aspects have been documented to suffer limited reuse [22], hard to predict behavior [33], and difficult modular reasoning [16][1]. All these factors hinder useful software engineering practices such as step-wise development [44][11] and its natural materialization in *component-based software engineering (CBSE)* [42], where programs are developed incrementally by composing components one at a time.

An aspect is a declaration of changes that are to be made to a program; the process of making these changes is called *weaving*. There have been several proposals to define aspect semantics [19]. The most common uses an event-based model [43]. There are many ways in which aspects and weaving can be implemented. The historical roots of AOP are in *meta-object protocols (MOPs)*[14], which implement weaving at run-time. A MOP does not change a program's source or binaries; it changes only a program's run-time execution. On the other hand, aspect compilers perform static weaving by producing woven binaries or woven source [4]. When a woven binary is run, its execution flow is indistinguishable from that of a MOP implementation. Aspect compilers are popular today because, among other reasons, they offer improved program run-time performance through static optimizations that are too expensive to realize via MOPs [5].

If we want to understand the impact that aspects have on a program's structure, we need to study an implementation of aspects that makes program structure explicit. This is where transformations come in. A *program transformation* is a function that maps programs to programs [34]. While there are few aspect compilers [21] that explicitly use program transformation tools [39], we know that the effects of static weaving can be understood in terms of transformations. This connection enables us to raise aspects from code artifacts to mathematical entities (functions from programs to programs) and develop algebraic models of aspects and their composition. These models reveal aspect composition as a significant source of the problems mentioned above. We propose an alternative model of aspect composition that eliminates these problems while preserving the power of AspectJ. We model advice weaving as function composition, which we have shown is essential to the synthesis of large-scale programs [9][10]. To the best of our knowledge, large-scale program synthesis has not been tackled yet with AOP technology [3]. We believe our model lays an algebraic foundation on which to build and understand AOP tools.

2 AspectJ Overview

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, that we illustrate and interpret as transformations.

2.1 Static Crosscuts

Static crosscuts affect the static structure of a program [7][27]. We focus on *introductions*, also known as *inter-type declarations*, that add fields, methods, and constructors to existing classes and interfaces. In AspectJ, standard Java classes and interfaces are referred to as *base code*. Consider class `Point` defined below:

```
class Point {
  int x;
  void setX(int v) { x = v; }
}                                                    (1)
```

The following aspect `TwoD` adds (introduces) a second coordinate value to class `Point`. It adds field `y` and method `setY`:

```
aspect TwoD {
  int Point.y;
  void Point.setY(int v) { y = v; }
}
```

When these two files are composed or *woven* by the AspectJ compiler `ajc` using the command:

```
ajc Point.java TwoD.java
```

The result is a new class `Point'` with the introduced members underlined below:

```
class Point' {
  int x;
  void setX(int v) { x = v; }
  int y;
  void setY(int v) { y = v; }
}                                                    (2)
```

AspectJ generally uses more sophisticated rewrites than those shown in this paper. The composed code snippets we present simplify illustration and are behaviorally equivalent to those produced by `ajc`.

Static Crosscuts as Transformations. From the program transformation perspective, base code such as `Point` in `(1)` represents a value to which a function (a program transformation) or aspect is applied. For instance, class `Point'` in `(2)` can be written as the following expression:

```
Point' = TwoD ( Point )
```

That is, `Point` is a base program and `TwoD` is a function that maps `Point` to `Point'`.

2.2 Dynamic Crosscuts

Dynamic crosscuts, in contrast, 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 [28][43]. 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.

The following aspect is the familiar logging example. Its interpretation is: run the advice code (underlined) `after` (advice type) the execution of methods in class `Point` whose name starts with '`set`' (*pointcut in italics*).

```
aspect Logging {
  after():  execution(* Point.set*(..))
  { println("Logged"); }
}                                                    (3)
```

From a compiler perspective, an equivalent interpretation is: *insert* the advice code after the body of any method in class `Point` whose name starts with '`set`'. For example, if aspect `Logging` is woven into class `Point'` in `(2)` the result is equivalent to:

```
class Point" {
  int x;
  void setX(int v) {x = v; println("Logged"); }
  int y;
  void setY(int v){ y = v; println("Logged"); }
}                                                    (4)
```

Dynamic Crosscuts as Transformations. Dynamic crosscuts can be implemented by transformations. For example, class `Point"` in `(4)` can be written as the expression:

```
Point" = Logging(Point') = Logging(TwoD(Point))
```

That is, class `Point"` is the result of applying two transformations, or from an AOP perspective the result of weaving two aspects, into class `Point`.

AspectJ provides an array of sophisticated mechanisms to define powerful pointcuts and to perform complex rewrites when weaving aspects into programs. All dynamic crosscuts can be understood as transformations including pointcut designators such as `cflow`, `args`, `this`, and `target`, which expose context information of a join point [7].

Consider `cflow(Y)` where `Y` is a pointcut. Suppose `Y` captures a specific method execution or method call. `cflow(Y)` is the set of join points that occur during the execution of `Y`, from the time that the method is called to the time of the return [7]. An interesting question to ask is if a join point `X` occurs within the control flow of `Y`? The pointcut that expresses this is concisely written in AspectJ as:

```
cflow( Y ) && pointcut_for_X
```

From a compiler's perspective, control flow advice is a transformation that is composed from four simple transformations: (i) introduce a control flow stack `S`, (ii) before each `Y` join point, push a marker `M` on `S`, and (iii) after each `Y` join point, pop `M` off `S`. For the duration that `M` is on `S`, any join point that occurs does so within the control flow of `Y`. And finally, (iv) at each `X` join point, check to see if `M` is on `S`; if so, execute the advice code.

Aspect compilers, such as `ajc`, demonstrate that aspects can be implemented by transformations: `ajc` takes a base program and aspects as input and produces a woven binary as output. Even so, the connection of dynamic crosscuts, especially `cflow`, to transformations remains controversial [23]. However, when given proper consideration, optimization and weaving techniques such as those presented in [5][24][32] are examples of program transformations, sophisticated indeed, but transformations nonetheless.

2.3 Advice Precedence

Recognizing that aspects can be realized as program transformations is a key first step in understanding how aspects impact program structure. The next step is to see how aspects are composed.

Multiple pieces of advice can be applied to the same join point. *Advice precedence* determines the order in which advice is woven. AspectJ deals with precedence differently depending on where the pieces of advice are defined, either in the same aspect or in different aspects [7].

Ordering aspects. AspectJ programmers have the option of declaring the order in which aspects are woven by a precedence statement such as:

```
declare precedence: Aspect3, Aspect2, Aspect1;
```

In the above example, the advice of $\texttt{Aspect}_1$ is woven first, then the advice of $\texttt{Aspect}_2$, and finally the advice of $\texttt{Aspect}_3$.[1] If no precedence statement is declared, the precedence of aspects is undefined in the semantics of AspectJ. In such cases, the AspectJ compiler chooses an order in which to weave aspects. In general, this order cannot be inferred by programmers prior to weaving.

Unfortunately, different weaving orders can result in different programs. Let us demonstrate with an analogy from arithmetic: given a value `2` and functions `double(x)`, `add3(x)`, and `sub2(x)` doubling the input, adding `3`, and subtracting `2`, respectively, different orders of their composition yields different results: `double(sub2(add3(2))=6` but `sub2(double(add3(2)))=8`. This is the reason why expressions, not sets of operations, are evaluated.

In our setting, a value is a base program and functions are aspects. We need to know the weaving order to be able to predict the result. If the weaving order is undetermined, as in the absence of a precedence declaration, the woven program will at the least be not portable (since different compilers can choose different weaving orders). Moreover, programmers will not be informed about the order the compiler chooses, i.e., they will find it hard to predict the result of the weaving.

It is clear that to use arithmetic effectively, we must consider the order of operations. We argue that the same applies to aspect weaving.

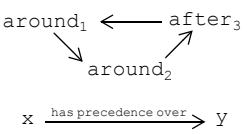
Ordering advice. The precedence of advice in an aspect is governed by the following rules copied verbatim from [7]:

> If two pieces of advice are defined in the same aspect, then there are two cases:
>
> • If either are after advice, then the one that appears later in the aspect has precedence over the one that appears earlier.
>
> • Otherwise, then the one that appears earlier in the aspect has precedence over the one that appears later.

In the program text, different pieces of advice are necessarily ordered. However, these precedence rules lead to two problems: 1) they may introduce a circularity such that the compiler cannot decide with which piece of advice to start the weaving, and 2) they cannot express all composition (weaving) orders.

The circularity problem is well-known [7], but the latter is not. A single aspect with three pieces of advice (identified by subscripts) illustrates both:

```
aspect Circular {
  void around1():execution(void test.main(..))
   { println("A1"); proceed(); println("A1"); }
  after3(): execution(void test.main(..))
   { println("A3"); }
  void around2():execution(void test.main(..))
   { println("A2"); proceed();println("A2"); }
}                                                    (5)
```



First, when the rules are applied, a circular precedence is created, as illustrated in the diagram to the right. To resolve the problem, programmers must manually modify the order in which the advice is listed in the program text, and ensure that the resulting weaving order eliminates circularity and produces a semantically appropriate weaving for the task at hand, a non-trivial and lengthy process.

Second, some composition orders cannot be attained. Suppose we want the following output sequence `(A2, A1, <main>, A1, A3, A2)`, which is achieved by weaving $\texttt{around}_1$ first, then $\texttt{after}_3$, and then $\texttt{around}_2$. In what order should advice $\texttt{around}_1$, $\texttt{after}_3$, and $\texttt{around}_2$ be listed in a single aspect file to achieve this weaving order?

The above rules dictate that $\texttt{around}_2$ must be listed before $\texttt{around}_1$ (because $\texttt{around}_1$ must be woven first). Advice $\texttt{after}_3$ must also appear before $\texttt{around}_2$ (to weave $\texttt{after}_3$ first). Thus, the ordering so far is: $\texttt{after}_3$ then $\texttt{around}_2$ then $\texttt{around}_1$. But $\texttt{after}_3$ must also appear after $\texttt{around}_1$ (for $\texttt{around}_1$ to be woven before $\texttt{after}_3$). It is impossible for $\texttt{after}_3$ to be *both* before $\texttt{around}_2$ and after $\texttt{around}_1$. Thus, no linear ordering of the advice $\texttt{around}_1$, $\texttt{around}_2$, and $\texttt{after}_3$ can achieve the desired weaving order.

A way to realize such a weaving is to store each advice in a separate aspect file and use `declare precedence`:

1. The mathematical concept of precedence has the opposite meaning of precedence in AspectJ. AspectJ precedence means apply last, whereas mathematical precedence means apply first.

```
declare precedence: around2, after3, around1;
```

Another way might be to convert all after and before advice into around advice, which can be easily ordered. But this then begs the question of why after and before advice have different ordering rules than around advice.

In summary, the current rules for precedence makes program reasoning unnecessarily difficult. But precedence is not the only problem with aspect composition. Fundamental software engineering practices such as step-wise development are not satisfactorily supported by AspectJ, as the following section shows.

3 An Incremental Development Example

Incremental or *step-wise development (SWD)* is a fundamental programming practice [10][42][44]. It aims at building complex programs from simpler ones by progressively adding programmatic details. SWD was not fully appreciated by the software engineering community for years. Today it is a centerpiece of core results in the synthesis of programs in product-lines [10] and component-based software engineering [42].

We illustrate a small but typical example of incremental development. We use subscripts to denote a particular version of our program at a given step and underline the code that is added by each increment.

Base. Class $\texttt{Point}_0$ defines a 1-dimensional point with an `x` coordinate and corresponding `setX` method:

```
class Point0 {
   int x;
   void setX(int v) { x = v; }
}                                                    (6)
```

First increment. Adds coordinate `y` and its `setY` method to $\texttt{Point}_0$. The result is:

```
class Point1 {
   int x;
   void setX(int v) { x = v; }
   int y;
   void setY(int v) { y = v; }
}
```

Second increment. Counts how many times the set methods are executed. Adding both increments to base yields:

```
class Point2 {
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int counter = 0;
}
```

Third increment. Adds a `color` field and its corresponding set method to $\texttt{Point}_2$:

```
class Point3 {
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int counter = 0;
   int color;
   void setColor(int c) { color = c; }
}
```

Now here is an implementation in AspectJ:

Base. Identical to `(6)` because classes are the base code of AspectJ applications.

First increment. We define aspect `TwoD` that introduces field `y` and method `setY` to class `Point`:

```
aspect TwoD {
   int Point.y;
   void Point.setY(int v) { y = v; }
}
```

The command that composes class $\texttt{Point}_0$ and aspect `TwoD` and achieves a program equivalent to $\texttt{Point}_1$ is:

```
ajc Point0.java TwoD.java
```

Second increment. Aspect `Counter` introduces field `counter` to class `Point` and advises the execution of all set methods to increment this counter[2]:

```
aspect Counter {
  int Point.counter = 0;
  after(Point p) : execution(* Point.set*(..))
                   && target(p)
  { p.counter++; }
}                                                    (7)
```

A program that is equivalent to $\texttt{Point}_2$ is produced by:

```
ajc Point0.java TwoD.java Counter.java
```

Third increment. Aspect `Color` adds a `color` field and a `setColor` method:

```
aspect Color {
 int Point.color;
 void Point.setColor(int c) { color = c; }
}
```

The composition of base with the three increments is:

```
ajc Point0.java TwoD.java Counter.java Color.java
```

However, this time the result is not $\texttt{Point}_3$, but instead:

```
class Point3' {
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int counter;
```

2. In AspectJ there are other ways to define `Counter`. Shortly, we will present the rationale behind this implementation decision.

```
   int color;
   void setColor(int c){ color=c; counter++; }
}
```

That is, the `setColor` method of $Point_3'$ increments `counter` (underlined above) unlike $Point_3$. This means that developers face the problem that building a program incrementally by hand may yield different results than using AspectJ. To produce $Point_3$ using AspectJ, we should have used a more constrained version of `Counter` that captures execution join points only of `setX` and `setY` methods:

```
aspect Counter {
  int Point.counter = 0;
  after(Point p) : (execution(* Point.setX(..))
                 || execution(* Point.setY(..)))
                 && target(p)
  { p.counter++; }
}                                                    (8)
```

An obvious question is: why was `Counter` not defined like `(8)` in the first place? Doing that certainly would solve *this* problem, but we must consider other properties of software modules that are also desirable for aspects. Among them is reusability, i.e., we want to treat aspects as components as in CBSE and reuse them *as is*. For example, suppose `Counter` is redefined as `(8)`, but now we want to build program $Point_3'$ instead. We would have to revise `Counter` back to `(7)` as the version in `(8)` cannot be used. The question is: why can we not reuse the same aspect for both cases? The problem is that aspect weaving does *not* distinguish among development stages of a program. We show how to solve this problem in the next section.

4 An Algebraic Model of Aspects

In this section, we develop an algebraic model that reveals another source of complexity in AspectJ composition. We then propose an alternative model of composition that retains the power of AspectJ, supports step-wise development, simplifies advice precedence, and facilitates reasoning using aspects. Our model has three operations that build upon the notions of introduction, advice, and weaving.

4.1 Preliminaries

Our model requires all method introductions to be explicit. Advice in AspectJ implicitly introduces a method. Recall the `Logging` aspect:

```
aspect Logging {
  after(): execution(* Point.set*(..))
  { println("Logged"); }
}
```

When woven into class $Point_0$ in `(6)`, aspect `Logging` can be regarded as the transformation that results in:

```
class Point  {
   int x;
   void setX(int v) { x = v; printLog(); }
   static void printLog(){ println("Logged");}
}
```

Method `printLog` is an explicit method that contains the advice body (the log message) and it is called at the end of the body of method `setX` (after the method execution).

To separate advice from introductions, *pure advice* is a named advice that replaces the advice body with a method call. To preserve AspectJ semantics, this call is not advisable, i.e., it has no join points. All join points of the original advice body reside in the method that is called. For example, we can conceptually rewrite the `Logging` aspect as:

```
aspect Logging {
  static void Point.printLog()
   { println("Logged"); }

  Log is after():execution(* Point.set*(..))
        --> Point.printLog();
}                                                    (9)
```

where `Log` is the name given to the pure advice, `printLog` is the method that contains the advice body, and the `-->` arrow indicates the method call. Other researchers have advocated a similar concept (that of making advice bodies into explicit methods), but have motivated the idea for different reasons [35]. Without loss of generality, we assume all advice is pure advice from this point on in this paper.

4.2 Introduction Sum

An *introduction* is a function that adds a data member or method to a program. Recall aspect `TwoD` and class $Point_0$ whose composition was modeled algebraically as:

$$Point_1 = TwoD(Point_0) \quad (10)$$

where $Point_0$ and $Point_1$ are values, and `TwoD` is a function that maps class $Point_0$ to class $Point_1$. Appealing to intuition, we can rewrite `(10)` as the sum of the introductions of `TwoD` with $Point_0$:

$$Point_1 = TwoD + Point_0 \quad (11)$$

Operation `+` is called *introduction sum*. It is a binary operation that performs disjoint union on *program fragments*, which are sets of variables and methods. For example, aspect `TwoD` is the program fragment (set) containing `y` and `setY`, and class $Point_0$ is the fragment (set) containing `x` and `setX`. We write introduction sum as:

$$\begin{aligned} Point_1 &= TwoD + Point_0 \\ &= (setY + y) + (setX + x) \\ &= setY + y + setX + x \end{aligned}$$

meaning $Point_1$ = {setY, y, setX, x} where our notation above omits set brackets. As + is disjoint set union, introduction sum has the following properties:

Identity. `0` is the *empty program* (i.e., a program fragment that contains no members). If `X` is a program fragment:

```
X = X + 0 = 0 + X
```

Commutativity. + is commutative because set union is commutative.

Associativity. + is associative as set union is associative.

+ differs from AspectJ introduction in that it does not allow member overriding. We believe overriding is rarely used and can be circumvented with a more structured design.[3]

4.3 Weaving

Pure advice is a function that maps an input program to a program where calls to advice methods have been inserted. We use function application to model the operation of *weaving*. Let `a` be pure advice and `P` be a program. The result of applying (weaving) `a` into `P` is program `P'`:

```
P' = a(P)
```

Weaving has the following properties:

Identity. `id` is the *null pure advice* — i.e., pure advice that captures no join points. Null pure advice is the identity transformation; its application does not affect a program. If `P` is a program fragment, `P=id(P)`. That is, `P` does not change when woven with `id`.

Associativity. Weaving is right associative.

Distributivity. Weaving distributes over introduction sum. Let `P` be a program, `a` be pure advice, and `P'=a(P)`. Suppose `P=X+Y+Z`, where `X`, `Y`, and `Z` are arbitrary program fragments. We have:

```
P'  = a(P)
    = a(X + Y + Z)
    = a(X) + a(Y) + a(Z)
```

Advice applies to *all* join points in a program. Thus it is immaterial if the program fragment is viewed as a whole (`P`) or as the sum of its parts (`X+Y+Z`). This distributivity property is central to AOP.

4.4 Advice Sum

Each piece of advice is a transform (i.e., a function). The application of multiple pieces of advice is modeled by function composition, denoted by •, which we call *advice sum*. • also models advice precedence. `a3•a1` means apply advice `a1` first and then `a3`. Advice sum has the properties:

Identity. `id` is the null pure advice. If `a` is pure advice:

```
a = a • id = id • a
```

Commutativity. The order in which advice is applied matters. • is not commutative.[4]

Associativity. • is associative because function composition is associative.

4.5 Modeling Aspects as Vectors

We model an aspect as a vector of two entries. The first entry, called the *advice part*, is the aspect's advice and the second entry, called the *introduction part*, is the aspect's introductions. The vector for `Logging (9)` is:

```
Logging = <Log, printLog>
```

where `Log` is the name of the pure advice and `printLog` is the name of the introduced method.

`Counter` is another example. A pure advice version of it is:

```
aspect Counter {
   CounterP is after(Point p):
     execution(* Point.set*(..)) && target(p)
     --> Point.counterInc(p);

   static void Point.counterInc(Point p)
     { p.counter++; }
   int Point.counter = 0;
}
```

and its vector is:

```
Counter = <CounterP, counterInc + counter>
```

Note that the second entry of the vector sums the introductions `counterInc` (the method of the advice body) and `counter` (the variable).

Finally, a pure version of the `Circular` aspect `(5)` is:

```
aspect Circular {
   pointcut pcd() : execution(void test.main(..));
   static void test.m1(){ ... }
   static void test.m3(){ ... }
   static void test.m2(){ ... }
   a1 is void around1(): pcd() --> test.m1();
   a3 is after3():       pcd() --> test.m3();
   a2 is void around2(): pcd() --> test.m2();
}
```

Suppose advice is woven in the textual order listed in an aspect. `Circular` would be modeled by the vector:

```
Circular = <a2 • a3 • a1, m2 + m3 + m1>
```

The expression `a2•a3•a1` represents compound advice, where `a1` is woven first and `a2` last.

4.6 Aspect Composition

Let aspects `A1` and `A2` be modeled by the vectors $A1=<a_1, i_1>$ and $A2=<a_2, i_2>$. We denote aspect composition

3. + also allows new classes and interfaces to be added to a program, which AspectJ does not support. This extra ability is useful when new functionality is added [30]. Aspects *can* encapsulate nested classes and nested interfaces, but *not* classes and interfaces that are unnested.

4. Two pieces of advice commute if they have no join point in common.

in AspectJ by operation ◊. The AspectJ composition of A2 with A1 (with A1 being applied first) is:

$$A2 \lozenge A1 = \langle a_2, i_2 \rangle \lozenge \langle a_1, i_1 \rangle = \langle a_2 \bullet a_1,\ i_2 + i_1 \rangle$$

◊ is similar to vector addition because the coordinates of vectors are summed: + sums program fragments, • sums advice in weaving order.

As another example, program P can be modeled by the vector <id,p>, where id is null pure advice and p is the introduction sum of the members of P. Weaving aspect A1 into P and then weaving aspect A2 is:

$$\begin{aligned} A2 \lozenge A1 \lozenge P &= \langle a_2, i_2 \rangle \lozenge \langle a_1, i_1 \rangle \lozenge \langle id, p \rangle \\ &= \langle a_2 \bullet a_1 \bullet id,\ i_2 + i_1 + p \rangle \\ &= \langle a_2 \bullet a_1,\ i_2 + i_1 + p \rangle \end{aligned}$$

The *code* of a vector is the program that the vector represents. Let V be a vector. Its code, denoted [V], is computed by weaving its advice part with its introduction part:

$$[V] = [\langle a, i \rangle] = a(i) \qquad (12)$$

This follows from the fact that advice can advise any join point of a program[5]. Thus the program that is produced by weaving A1 and then A2 into P:

$$[A2 \lozenge A1 \lozenge P] = a_2 \bullet a_1(i_2 + i_1 + p) \qquad (13)$$

More generally, when aspects $A_1 \ldots A_n$ are woven in this order into P, the result is:

$$[A_n \lozenge A_{n-1} \lozenge \ldots \lozenge A_1 \lozenge P] = a_n \bullet a_{n-1} \bullet \ldots a_1(i_n + i_{n-1} + \ldots + i_1 + p) \qquad (14)$$

That is, the result of weaving a sequence of aspects into a program equals the weaving of advice in weaving order into the program that is the introduction sum of the program's members and aspect introductions. (14) represents the "shape" of any program produced by AspectJ. We call this the *vector model* of composition.

(14) identifies the source of the problems noted earlier in Section 3 about incremental program development using AspectJ. It can be seen in the expansion of (13):

$$\begin{aligned} [A2 \lozenge A1 \lozenge P] &= a_2 \bullet a_1(i_2 + i_1 + p) \\ &= a_2 \bullet \underline{a_1}(i_2) + a_2 \bullet a_1(i) + a_2 \bullet a_1(p) \end{aligned}$$

The offending term is underlined. It means that to apply aspect A2, the programmer is required to know how an advice from a previous development step (a_1) affects an introduction added in the current step (i_2). More generally, the weavings that cause problems in incremental development are underlined below:

$$\ldots \bullet a_{k+2} \bullet a_{k+1} \bullet a_k \bullet \underline{a_{k-1}} \bullet \underline{\ldots} \bullet \underline{a_2} \bullet \underline{a_1}(i_k) + \ldots$$

In other words, a programmer needs to know how previously applied pieces of pure advice a_j affect later introductions i_k where $j < k$. These are the terms that make step-wise development difficult. The problem is aggravated when a large number of aspects are composed and the development involves multiple steps.

4.7 The Functional Model

An alternative way to compose aspects is to equate aspect composition with function composition. Consider aspect A=<a,i>. We can model A as the function:

$$A(x) = a(i + x)$$

That is, A adds its introductions (i) to its program fragment input (x) before weaving its advice (a). So applying aspect A1 to program P and then applying aspect A2 is:

$$\begin{aligned} A2(A1(P)) &= a_2(i_2 + a_1(i_1 + p)) \\ &= a_2(i_2) + a_2 \bullet a_1(i_1) + a_2 \bullet a_1(p) \end{aligned} \qquad (15)$$

Note that the offending pure advice a_1 disappears from the left summand ($a_2(i_2)$). This generalizes to the composition of any number of aspects: the weavings that make step-wise development difficult are never generated. We call this the *functional model*. In effect, the vector model expresses *unbounded* quantification (i.e., the scope of advice extends over the entire program) [20], whereas the functional model expresses *bounded* quantification (i.e., the scope of advice extends over a particular stage in a program's development). An obvious question arises: which model is more expressive?

The functional model can express programs that the vector model cannot express, for example program (15). This expression cannot be produced by the vector model because all pieces of advice are woven into all introductions yielding program (13). Again, this is a consequence of the unbounded quantification of the vector model.

Unbounded quantification is a special case of bounded quantification. Think of quantifiers in first-order logic: the scope of a quantifier extends from its position in a logical formula to the right. Unbounded quantification means that the formula starts with the quantifier. In the functional model, a weaving expression is interpreted the same way: the influence of advice extends to the right and not to the left. Let us illustrate this with program (13) which requires unbounded quantification. To construct this program in the functional model, we map aspect A1 to a pair of aspects, one containing introduction i_1 and the other with a_1:

$$\begin{aligned} A1_{intro}(x) &= i_1 + x \\ A1_{advice}(x) &= a_1(x) \end{aligned}$$

Similarly for aspect A2:

5. Readers familiar with advice that advises itself will recognize that the pure advice part advises its introduction part. This is modeled by (12).

$A2_{intro}(x) = i_2 + x$
$A2_{advice}(x) = a_2(x)$

To build (13), we weave the aspects with introductions to P first and then the aspects with advice to get:

$A2_{advice}(A1_{advice}(A2_{intro}(A1_{intro}(P))))$
$= a_2 \bullet a_1(i_2 + i_1 + p)$

This ability to model unbounded quantification in the functional model is general, and is not specific to this particular example.

In summary, the functional model can express all programs that the vector model can express, and more once aspects are expressed in terms of bounded quantification. There are many useful programs that need bounded quantification. Recall the Point example of Section 3. Let us add a third dimension to Point, which is defined by aspect ThreeD that introduces a z variable and setZ method. Assuming that Counter advises all set methods as in (7) using bounded quantification, we can build at least four programs:

(a) Color(ThreeD(TwoD(Counter($Point_0$))))
(b) Color(ThreeD(Counter(TwoD($Point_0$))))
(c) Color(Counter(ThreeD(TwoD($Point_0$))))
(d) Counter(Color(ThreeD(TwoD($Point_0$))))

(a) is a program that counts the executions of setX. (b) counts the executions of setX and setY. (c) counts the executions of setX, setY, and setZ. (d) counts the execution of all set methods. Each of these programs is synthesized by reusing and composing aspects *as is*. Using AspectJ, weaving aspects Counter, Color, ThreeD, and TwoD in an arbitrary order into P will always produce program (d). To build all four programs using AspectJ would require four different versions of Counter.

To summarize, problems in step-wise development arise using AspectJ when pointcuts are not bounded to a set of classes, methods, and variables at a specific stage of program development. Common examples are pointcuts that capture the set of all calls to one or more methods, and wildcard patterns. Subsequent introductions that are captured by these pointcuts give rise to the problems discussed here. The functional model avoids these problems.

5 Perspective

We showed in Section 2.3 that precedence in AspectJ makes reasoning about and composing programs unnecessarily difficult. We showed in Section 4.7 that when aspects are reused as is, many programs cannot be built. We will show how to remove these limitations in Section 5.2 without sacrificing the power of AspectJ. Further, our work places aspects closer to key results in automated software design, which we now discuss.

5.1 Automated Software Design

The history of automated software design is replete with results on transformations and their connection to program structure. Tool-enabled program refactorings are program transformations [41]. Layers in layered software designs are transformations [9]. When viewed as increments in program functionality, features in software product-lines are transformations [10]. Model transformations play a key role in Model Driven Architectures; they are mappings between models (which are non-code representations of programs) [12]. Arguably the most significant result in automated software design is relational query processing [38]: a query evaluation program is defined by a composition of relational algebra operations. These operations are transformations of query evaluation programs.

Aspects define very useful transformations, and their strength is that programmers do not need understand transformation technologies to use them. As mentioned earlier, transformations are just one of a number of ways in which aspects can be implemented. The advantage of viewing aspects as transformations is that it exposes how aspects modify a program's structure. Doing so places aspects in context with results in software architectures and automated software development that also define and modify program structure.

5.2 Impact On Existing Tools

Our work eliminates several problems in AspectJ.

First, the precedence rules for ordering pieces of advice within an aspect (Section 2.3) can be eliminated. We propose a simpler rule: apply advice in the order in which it is listed in an aspect file. This rule will simplify the ordering algorithms currently utilized by aspect compilers and will help AspectJ programmers by reducing the effort to determine a composition order.

Second, the rules that AspectJ uses to assign precedence to aspect files can also be eliminated. We propose that a precedence be declared for *all* aspect files to define their composition order. Alternatively, the compiler could raise an error when users fail to specify an order where ordering matters. Again, this change simplifies advice ordering algorithms for multiple aspect files and also helps programmers as now aspect compiler output will be predictable.

AOP researchers have raised the issue that it should be unnecessary to specify a composition order when aspects are provably commutative. We agree. In cases where pointcuts have disjoint sets of join points their corresponding advice is commutative. Existing tool support can help identify these situations [7]. However, it is still necessary to specify *when* these pieces of advice are to be applied. This may require an enhancement of existing tools.

5.3 Related Work

Since the Sixties, the paradigm of layered software has been applied to harness the complexity of large software systems (initially, they were operating systems [18]). A key property of layered software is that a layer has knowledge of lower layers but not of higher layers. This is bounded quantification. We apply this principle to aspect orientation: the weaving of an advice affects the existing program, but not any parts that are added later.

Compositional models of aspects have a long history. GenVoca, AHEAD, and HyperJ are examples [9][10][20][40]. Relating compositional models to algebras is discussed in [10]. An early version of the ideas in this paper were presented at an AOSD 2005 workshop [31].

The relationship between program transformations and aspects is not new. Lämmel studied the implementation of aspects as programs transformations [28]. Kniesel et al. developed `JMangler`, a backend tool to support AOP that relies in transformations at the bytecode level [26]. We extend these ideas by showing how a transformation view leads to an algebraic model of aspect composition.

McEachen and Alexander consider the problems caused by weaving bytecode that already contains woven aspects [33]. A *foreign aspect* is an aspect that has been woven; the woven bytecode is later imported by a third party that has no access to the aspect's source code. A foreign aspect "comes alive" and can potentially affect subsequently added base or aspect code. Foreign aspects are problematic as they can: a) not capture all intended join points, b) capture unintended join points, and c) inadvertently interact with other aspects. The authors advocate guidelines to design the scope of pointcuts, use of abstract pointcuts to control the set of join points advisable by foreign aspects, and promote adequate pointcut documentation. Our work provides a foundation to understand the problems caused by foreign aspects and a solution to eliminate them. The rules of the functional model can be enforced by a compiler, whereas adhering to the guidelines of McEachen and Alexander is the responsibility of programmers.

Modular reasoning with AOP is controversial [15][16][17]. Kiczales and Mezini claim that in the presence of aspects "the complete interface of a module can only be determined once the complete configuration of modules in the system is known" [25]. In other words, aspects entail global reasoning that they define as "having to examine all the modules in the system or subsystems". The vector model of AspectJ mathematically corroborates their claim and shows the negative implications it has for incremental development. The functional model of composition reduces the need of global reasoning without restricting the power of AspectJ.

Rinard et al. propose a framework to classify aspects based on their interactions with other aspects and base code [36]. They present a tool that alerts users of cases where modular reasoning (a user-defined property) could be compromised so that users can take corrective action when necessary. *Open modules* proposes a module system whereby an interface describes the pointcuts and join points that are advisable by the pieces of advice of other modules thus promoting modular reasoning about aspects [1].

Complementary to our approach, there is work that aims to improve aspect reuse by making significant language changes. Gybels and Brichau propose a logic-based crosscut language to better decouple aspects from programs [22]. Rho and Kniesel propose aspect uniform genericity, application of logic metavariables in language constructs, as a way to promote reuse and to significantly expand the generic capabilities of AspectJ [37]. Incidentally, they too take a transformation view of aspects.

Classpects are an attempt at unifying aspects and classes [35]. Classpects are classes enhanced with bindings. A *binding* associates an advice type (`before`, `after`, `around`) and a pointcut with a call to a list of methods. These methods replace the advice body, similar to what we did when we transformed advice into pure advice.

A full implementation of the functional model requires advances in compilers. One issue is separate compilation [13][2], which is gaining attention in the programming languages community. Our need for separate compilation comes from the commutativity of operation `+`, which permits the introduction of a method before introducing other members on which the former may depend. This problem exists now in AspectJ because aspect files cannot be compiled separately from their base program.

6 Conclusions

Aspect-oriented programming should be in the repertoire of tools and techniques used by software developers. But the current model and flagship tool of AOP, AspectJ, has limitations: aspect reuse is hard, woven programs can have hard to predict behavior, modular reasoning using aspects is difficult, and step-wise development of programs is error-prone. We explored these limitations and found that a significant source of complexity in AspectJ is its model of aspect composition.

To address these problems, we recognized that there are many ways in which aspects could be implemented. We selected a way — using program transformations — that revealed how aspects alter a program's structure. This allowed us to raise aspects from code artifacts to mathematical entities (functions that transform programs) and enabled us to develop an algebra to model aspect composi-

tion. Our algebra exposes a source of the current problems, it also reveals a solution. By equating aspect composition with function composition, the problems were eliminated and the power of AspectJ was preserved.

We are now investigating how to include other AspectJ capabilities such as `declare parents`, abstract aspects, abstract pointcuts, and aspect inheritance into our model. Our goal is to build languages and tools based on our model and to evaluate their potential in an experimental setting using the *Aspect Bench Compiler (`abc`)* [8][4]. We believe that our work lays an algebraic foundation on which to build and understand AOP tools.

Acknowledgements. We thank Gary Leavens, Oege de Moor, Axel Rauschmayer, Jim Cordy, and Dewayne Perry for their comments on drafts of this paper.

This research is sponsored in part by NSF's Science of Design Project #CCF-0438786.

7 References

[1] J. Aldrich. Open Modules: Modular Reasoning about Advice. *ECOOP 2005*.

[2] D. Ancona, G. Lagorio, and E. Zucca, "True Separate Compilation of Java Classes", *PPDP 2002*.

[3] AOSD Europe Network of Excellence Workshop. *ECOOP* 2005.

[4] P. Avgustinov, et al., "abc: An Extensible AspectJ Compiler", *AOSD 2005*, Chicago, USA.

[5] P. Avgustinov, et al. "Optimizing AspectJ", *PLDI 2005*.

[6] AspectJ, version 1.2.1, `eclipse.org/aspectj/`.

[7] AspectJ Manual, `www.eclipse.org/aspectj/doc/progguide/language.html`.

[8] Aspect Bench Compiler. `www.aspectbench.org`

[9] D. Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

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

[11] M. Broy and E. Denert, *Software Pioneers — Contributions to Software Engineering*, Springer-Verlag, 149-169, 2002.

[12] J. Bezivin, "From Object Composition to Model Transformation with the MDA", *TOOLS'USA*, August 2001.

[13] L. Cardelli, "Program Fragments, Linking, and Modularization", *POPL 97*.

[14] S. Chiba, "Program Transformation with Reflective and Aspect-Oriented Programming", in [29].

[15] C. Clifton and G. Leavens, "Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning", *FOAL* 2002.

[16] C. Clifton, G.T. Leavens. "Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy". SPLAT *2003*.

[17] C. Clifton, "A Design Discipline and Language Features for Modular Reasoning in Aspect-Oriented Programs", Ph.D. Dept. Computer Science, Iowa State, 2005.

[18] E.W. Dijkstra. "The Structure of the 'THE'-Multiprogramming System", *CACM*, May 1968.

[19] R. Dounce, D. Le Botlan. "Towards a Taxonomy of AOP Semantics". AOSD-Europe. Technical Report, July 2005.

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

[21] J. Gray et al. "A Technique for Constructing Aspect Weavers Using a Program Transformation Engine". *AOSD 2004*.

[22] K. Gybels and J. Brichau, "Arranging Language Features for More Robust Pattern-based crosscuts", *AOSD 2003*.

[23] K Gybels and K. Ostermann, discussions at *SPLAT 2005*.

[24] E. Hilsdale and J. Hugunin. "Advice weaving in AspectJ". *AOSD 2004*.

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

[26] G. Kniesel, et al. "JMangler - A Framework for Load-Time Transformation of Java Class Files". *SCAM 2001*.

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

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

[29] R. Lämmel, J. Saraiva, and J. Visser (Eds), *Generative and Transformational Techniques in Software Engineering*, 2005.

[30] R.E. Lopez-Herrejon, et al. "Evaluating Support for Features in Advanced Modularization Techniques". *ECOOP 2005*.

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

[32] H. Masuhara, G. Kiczales, "Modeling Crosscutting Aspect-Oriented Mechanisms". *ECOOP 2003*.

[33] M. McEachen, R.T. Alexander. "Distributing Classes with Woven Concerns - An Exploration of Potential Fault Scenarios". *AOSD 2005*.

[34] Partsch, H., Steinbrüggen, R.: Program Transformation Systems. *ACM Computing Surveys*, September (1983).

[35] H. Rajan, K.J. Sullivan, "Classpects: Unifying Aspect- and Object-Oriented Programming", *ICSE 2005*.

[36] M. Rinard, A. Salcianu, S. Bugrara. "A Classification System and Analysis for Aspect-Oriented Programs", *FSE 2004*.

[37] T. Rho, G. Kniesel. "LogicAJ - A Uniformly Generic Aspect Language." Submitted.

[38] P. Selinger, et al, "Access Path Selection in a Relational Database System", *ACM SIGMOD 1979*, 23-34.

[39] Semantic Designs. `www.semdesigns.com/`

[40] P. Tarr, H. Ossher, et al., "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *ICSE 1999*.

[41] L. Tokuda and D. Batory. "Evolving Object-Oriented Designs with Refactorings" *J. Automated Soft. Engr.* 8, 2001.

[42] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 2002.

[43] M. Wand, et al., "A Semantics for Advice and Dynamic Join Points in Aspect Oriented Programming", *TOPLAS 2004*.

[44] N. Wirth, "Program Development by Stepwise Refinement", *CACM* 14 #4, 221-227, 1971. Also in [11].