Software Engineering Properties and Languages for Aspect Technologies (SPLAT), March 2005.

Improving Incremental Development in AspectJ by Bounding Quantification

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

Abstract

Incremental software development is a process of building complex programs from simple ones by successively adding programmatic details. It is an effective and common design practice that helps control program complexity. However, incrementally building software using aspects can be more challenging than using traditional modules. AspectJ quantification mechanisms do not distinguish different developmental stages, and thus pointcuts can capture join points from later stages that they originally were not intended to advise.

In this paper we present an algebraic model to describe aspects and their composition in AspectJ. We show that the way AspectJ's composes aspects plays a significant role in this problem. We propose an alternative model to compose aspects that improves the support for incremental development. It bounds the scope of quantification and still preserves the power of AspectJ. We also show how bounded quantification contributes to aspect reuse.

1 Introduction

Incremental software development is a fundamental and common programming practice. It aims at building complex programs from simpler ones by adding programmatic details. Traditional modularization techniques support incremental development and rely on modular reasoning for this task. Aspects are a powerful mechanism for modularizing concerns that would otherwise be scattered and tangled with the implementation of other concerns. However, incrementally building software using aspects can be a more challenging task than using conventional modules as aspects usually cut across several module boundaries.

Thus, it may seem that the complexity in incremental software development comes from the need of *global reasoning*, knowing the implementation details of all the modules in a system and their relations, that crosscutting concerns entail [7]. There exist many tools that can ameliorate global reasoning [3][4][5][9]. They provide different browser and query capabilities to identify, modularize and understand concerns.

In this paper, we present an algebraic model of AspectJ aspects and their composition. Using this model, we show that, besides global reasoning, aspect composition is also a contributing factor to the complexity of incremental development using aspects. We propose an alternative model of aspect composition to reduce this complexity by bounding the scope of quantification while preserving the full power of aspects. We also show how bounded quantification contributes to aspect reuse.

2 Incremental Development Example

Consider the incremental development of a class that represent points in a graphics application.

Step1. Class `Point` is the following Java file which defines a one dimensional point (i.e., it has a single coordinate):

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

Step 2. A `Y` coordinate and its corresponding set method is added to `Point` with an aspect like `TwoD`:

```
aspect TwoD {
   int Point.y;
   void Point.setY(int y) { this.y = y; }
}                                                    (2)
```

Using AspectJ, the composition (weaving) of these two files using the command:

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

yields an augmented definition class `Point` below (we use subscripts to denote the version of `Point` at a given step):[1]

```
class Point2 {
   int x;
   void setX(int x) { this.x = x; }
   int y;
   void setY(int y) { this.y = y; }
}                                                    (3)
```

Step 3. The next step is to count how many times the set methods that modify the coordinates of a point are executed. If we were to do this increment manually, we would need to

1. AspectJ uses more sophisticated rewrites that those shown here. The resulting composed code snippets that we present illustrate the observable behavior but not the actual output of `ajc`.

add the underlined code and thus the composed class `Point` becomes:

```
class Point3 {
   int counter = 0;
   int x;
   void setX(int x) { this.x = x; counter++;}
   int y;
   void setY(int y) { this.y = y; counter++;}
}                                                    (4)
```

Alternatively, we can use aspects to modularize this change:

```
aspect Counter {
  int Point.counter = 0;
  after(Point p) : execution(* Point.set*(..))
                   && target(p) {
         p.counter++;
  }
}                                                    (5)
```

And then weave `Point` and `TwoD` using the following command to yield a result functionally equivalent to `(4)`:

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

Step 4. It is now required that class `Point` contains `color` information. This entails adding a member and a new set method, the additions are underlined below:

```
class Point4 {
   int x;
   void setX(int x) { this.x = x; counter++;}
   int y;
   void setY(int y) { this.y = y; counter++; }
   int counter;
   int color =0;
   void setColor(int c) { this.color = c; }
}                                                    (6)
```

Again this step can be expressed using an aspect such as:

```
aspect Color {
 int Point.color =0;
 void Point.setColor(int c) { this.color = c; }
}                                                    (7)
```

However, when woven with previous steps as in the following command line:

```
ajc Point.java TwoD.java Counter.java
    Color.java
```

The outcome is not that of `(6)` and is shown below as `Point'4`. The difference is in method `setColor` which also increments `counter`:

```
class Point'4 {
   int counter = 0;
   int x;
   void setX(int x) { this.x = x; counter++;}
   int y;
   void setY(int y) { this.y = y; counter++; }
   int color = 0;
   void setColor(int c){
     this.color=c; counter++;
   }
}                                                    (8)
```

Thus programmers face the paradox that building software incrementally using AspectJ and manually can yield different results. To get the same effect of `(6)` using AspectJ, we should have used a more constrained version of `Counter` that advises only the `setX` and `setY` methods:

```
aspect Counter {
  int Point.counter = 0;
  after(Point p) : (execution(* Point.setX(..))
                 || execution(* Point.setY(..)))
              && target(p) {
        p.counter++;
  }
}                                                    (9)
```

A key principle of incremental software development is that each step builds on previous steps. Thus a later step is not expected to require invasive changes of earlier steps to work correctly.

The problem is that AspectJ quantification cannot distinguish among developmental stages. In other words, pointcuts can advise join points that by incremental design they should not be advising. To solve this problem, we propose to bound the scope of quantification so that a pointcut can advise joint points exclusively from the current and earlier development steps. If quantification is bounded this way, aspect `Counter` would not need to change.

An obvious question readers could have in mind is: why was `Counter` not defined like `(9)` in the first place? 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, thus we want to treat aspects as modules and reuse them as is. For example, suppose `Counter` is defined as `(9)`, but now we want to build program `(8)` instead. We would have to revise `Counter` back to `(5)` as the version in `(9)` cannot be reused. Later in Section 7 we elaborate more on how bounded quantification enhances aspect reusability.

3 An Algebraic Model

We develop an algebraic model that reifies the intuitively clear distinction between the concepts of *inter-type declarations* (a.k.a. introduction) and pointcuts and advice in AspectJ [3].

Throughout this paper, we refer to the pair of a pointcut and the advice that uses it as a *PCA*. We present first a model of inter-type declarations.

3.1 A Model of Inter-Type Declarations

An aspect with an *inter-type declaration (ITD)* can be understood as a function that maps an input program to an output program. For instance, consider aspect `TwoD` in `(2)` when woven with class `Point` of `(1)`. We can model this composition algebraically as:

$$Point_2 = TwoD(\ Point\ ) \qquad (10)$$

where `Point` is a value[2], `TwoD` is a function that maps a program with a `Point` class to a program that has the augmented class $Point_2$. In `(10)`, the input program is class `Point`, and the output program is class $Point_2$.

Aspect `TwoD` is effectively adding (by means of ITDs) an instance field and a set method to class `Point`. Thus, appealing to intuition, we can rewrite `(10)` as a summation:

$$Point_2 = TwoD + Point \qquad (11)$$

We call operation + *concern addition*. It takes two arguments, which can be either a program or an ITD. Let P_1 and P_2 be programs that modularize disjoint sets of classes, and I_1 and I_2 be aspect files that contain ITDs that advise programs P_1 and P_2, respectively. The possibilities of the semantics of concern addition are:

- *Addition of two programs* P_1+P_2 yields the union of the classes of both programs.
- *Addition of two ITDs* I_1+I_2 yields a single aspect file formed by the union of the contents of both I_1 and I_2.
- *Addition of ITD and matching program* I_1+P_1 yields the classes of P_1 with their I_1 introductions.
- *Addition of ITD and non-matching program* P_2+I_1 or I_1+P_2 yields I_1 and the classes of P_2.

From the semantics of AspectJ's ITDs, we can deduce several properties of +.

Identity. Concern addition has an identity that we denote by `0`. If `X` is a program or an aspect file:

$$X = X + 0 = 0 + X \qquad (12)$$

In terms of code, `0` corresponds to an *empty program* (i.e., a program that contains no classes).

Commutativity. Concern addition is commutative. The order in which terms are added does not matter. This is consistent with AspectJ in that no variable or method (defined either in a class or ITD) can be introduced more than once[3].

Associativity. Let `A`, `B`, and `C` be programs or aspects with ITDs. To prove associativity, we must show that the following holds:

```
(A + B) + C = A + (B + C)
```

If we regard operation + as a form of weaving mechanism, an expression like `(A + B) + C` means: first weave `A` and `B` to get a result and then weave that result with `C`. The proof of this property follows from a case analysis of the four possibilities of concern addition shown above. The details of the proof are presented in the Appendix. What is important to note though is that for AspectJ this property does *not* hold. The reason is that the second and fourth cases of concern addition (addition of two ITDs and the addition of ITDs to a non-matching program) are not supported by AspectJ's weaver. Although our model is more general than AspectJ, this is not crucial for the results later in this paper.

Composition. `TwoD` is a composite inter-type declaration. It modularizes the addition of both the `setY` method and the `y` field. We can express `TwoD` algebraically as the sum of primitive inter-type declarations:

$$TwoD = setY + y \qquad (13)$$

Which we could substitute into `(11)` to produce an equivalent definition of $Point_2$:

$$Point_2 = setY + y + Point$$

3.2 A Model of PCAs

Consider the following aspect that consists of a single PCA and displays a log message after execution of set methods of class `Point`:

```
aspect Log {
 pointcut logP():execution(* Point.set*(..));
 after() : logP() {
    System.out.println("set called");
  }
}                                                    (14)
```

Advice code can be regarded as an implicit method declaration and call. Our model of PCAs reifies the body of an advice into an ITD method and a call to it. We also give an explicit name to each PCA. Conceptually aspect `Log` in `(14)` could be re-written as:

```
aspect Log {
  static void Point.setCalled() {
    System.out.println("set called");
  }
  LogP is after():execution(* Point.set*(..))
      --> Point.setCalled();
}                                                    (15)
```

2. We can regard `Point` as a string that represent the class code.

3. Using `declare precedence` statements, AspectJ can override ITDs [8]. We do not consider such cases in our current model.

where LogP is the name given to the PCA. Of course, (15) does not correspond to AspectJ syntax as advice is defined by a method *body* not a method *call*. For this reason, we call LogP declaration in (15) a *pure PCA*. Soon we will see why this distinction is important and the underlying reason behind it.

We can model aspect Log as a vector of two entries. The first contains the pure PCA (LogP) and the second the *reified advice* setCalled:

```
Log = [LogP, setCalled]                                    (16)
```

As another example, recall aspect Counter of (5). A "reified" version of that aspect could be:

```
aspect Counter {
   CounterP is after(Point p):
        execution(* Point.set*(..)) &&
        target(p) --> Point.counterA(p);
   static void Point.counterA(Point p){
        p.counter++;
   }
   int Point.counter = 0;
}                                                          (17)
```

This aspect can be modeled as the vector:

```
Counter = [CounterP, counterA + counter]                   (18)
```

where counter is the term that adds variable counter to class Point, similar to variable y in (13). Notice that the second entry in the vector composes ITDs.

We want to express the idea of the application of a pure PCA to a program. To do this, we need another operation different from concern addition.

3.3 Concern Multiplication

We call operation * *concern multiplication*. It takes two arguments, which can either be a PCA or a program. Let X_1, and X_2 be pure PCAs and Y a program. The possibilities of the semantics of concern multiplication are:

- *Multiplication of two PCAs* X_1*X_2 yields an aspect file that contains both PCAs.
- *Multiplication of a PCA and a program* X_1*Y yields a program with the PCA X_1 woven into program Y.

Suppose aspect A is the vector [m,a]. The result of applying A to P is:

```
P'   = A( P )
     = m * (a + P)                                         (19)
```

That is, A's ITDs are added to P, and then the pure PCA m is applied to the resulting program. Equation (19) follows directly from AspectJ observable semantics. In particular, a is added to P prior to multiplication because in AspectJ, advice can advise itself.

As in the case of concern addition, we can deduce several properties of *.

Identity. Concern multiplication has an identity that we denote by 1. In terms of code, 1 corresponds to the *null PCA*, the PCA that does not capture any join points. If P is a program and m is a pure PCA:

```
P = 1*P
m = m * 1 = 1 * m                                          (20)
```

Non-Commutativity. Concern multiplication is, in general, non-commutative. Applying advice in different orders can change the semantics of a program. Commutativity holds only in the case where the set of join points of different PCAs are disjoint.

Right-Associativity. In AspectJ, the order in which advice is applied is generally undefined. This means that a programmer cannot know a priori[4] the advice order simply by looking at the pointcut and advice code. We model the precedence of PCA application by the order in which PCAs are applied. If m_1 and m_2 are PCAs, m_2*m_1 means apply m_1 first and then apply m_2. This entails that * is right-associative.

Distributivity. Concern multiplication is distributive over concern addition. This follows from AspectJ observable semantics as well. Let P be a program and m be a pure PCA, and program P' = m*P. Now suppose P = A+B+C, where A, B, C are arbitrary Java files or ITDs. We have:

```
P'   = m * P
     = m * (A + B + C)
     = m*A + m*B + m*C
```

Operator * distributes over + because the right-hand argument of * defines the base code over which the PCA is quantified. So if the base code is partitioned, the quantification of the pointcut applies to each partition.

4 Composing Aspects

Given the above denotation of an aspect, we refer to the first entry of the vector as the aspect *multiplicative part* (denoted with m) and to the second entry as the *additive part* (denoted with a).

Let A_1 and A_2 be aspects, where A_1 = [m_1,a_1] and A_2 = [m_2,a_2]. The composition of A_1 and A_2, denoted by A_2•A_1, is similar to vector addition:

$$A_2 \bullet A_1 = [\ m_2 * m_1,\ a_2 + a_1\ ] \qquad (21)$$

That is, A_2•A_1 means apply A_1 first and then A_2. Thus A_1 has precedence over A_2. The result is equivalent to summing

4. There are special rules that apply for certain types of advice when advices are defined in either the same aspect or in others[3]. These rules can help determine the order in few cases but not in general. Additionally, declare precedence clauses can be used to enforce advice order.

their additive parts, and multiplying their pure PCAs in precedence order, which we assume (without loss of generality) to be in composition order.

We can model a base program P (a program without aspects) as the vector [1,P]. The application of A_1 and A_2 to P is:

$$\begin{aligned} A_2 \bullet A_1 \bullet P &= [m_2,a_2] \bullet [m_1,a_1] \bullet [1,P] \\ &= [\ m_2 * m_1 * 1,\ a_2 + a_1 + P\ ] \\ &= [\ m_2 * m_1,\ a_2 + a_1 + P\ ] \end{aligned}$$

Still using vector arithmetic concepts, the result of a weaving (i.e., aspects applied to a base program) is the *length* of the resulting vector V, which we denote by |V| and is computed by:

$$\begin{aligned} |V| &= |[m,a]| \\ &= m * a \end{aligned} \qquad (22)$$

Thus, the length of $A_2 \bullet A_1 \bullet P$ is:

$$|A_2 \bullet A_1 \bullet P| = (m_2 * m_1) * (a_2 + a_1 + P) \qquad (23)$$

More generally, the weaving of aspects $A_1 \ldots A_n$ into program P yields the augmented program:

$$|A_n \bullet A_{n-1} \bullet \ldots \bullet A_1 \bullet P| = (m_n * m_{n-1} * \ldots * m_1) * (a_n + a_{n-1} + \ldots + a_1 + P) \qquad (24)$$

This equation also follows directly from the observable semantics of AspectJ weaving.

5 Difficulty with AspectJ

A difficulty of incremental development using AspectJ is determining at each step the extent of the set of join points defined by a pointcut. Stated differently, *the programmer needs to know if any of the pointcuts applied in previous development steps affects the code added by current step or later steps.*

We can see this difficulty by expanding equation (23) using the distributivity of * over +:

$$\ldots = m_2 * \underline{m_1} * a_2 + m_2 * m_1 * a_1 + m_2 * m_1 * P \qquad (25)$$

The underlined term m_1 indicates that applying A_2 to a program requires the designer to know how m_1 affects a_2. This problem is aggravated when the number of aspects composed is large. Expanding (24), these are the products that cause problems in incremental development:

$$\ldots * m_{i+2} * m_{i+1} * m_i * \underline{m_{i-1}} * \underline{\ldots} * \underline{m_2} * \underline{m_1} * a_i + \ldots$$

That is, a designer is required to know how *previously* applied multipliers m_j affect subsequently added terms a_i, where i>j.

For instance, in our example of Section 2, when developing aspect Color it is necessary to check if previous pointcuts affect it. In this case, aspect Counter, applied in the previous step, contains a pointcut that causes method setColor to be advised and thus to incorrectly increment the counter.

We can express algebraically the example of Section 2 to verify this issue:

```
Point = [1, setX + x]
TwoD = [1, setY + y]
Counter = [CounterP, counterA + counter]
Color = [1, setColor + color]                    (26)
```

And the weaving can be expressed in the following way (see the Appendix for details):

```
|Color • Counter • TwoD • Point|
= counterP*setColor + color + counterA +
  counter + counterP*setY + y +
  counterP*setX + x                              (27)
```

The first element of the summation exposes the above-mentioned difficulty for incremental development.

6 An Alternative Composition Model

We propose an alternative model of aspect composition that supports incremental development while retaining the power of AspectJ. The idea is simply to treat aspect composition as function composition where quantification is bounded to its current input. Recall equation (19). Starting with the case when aspect A = [m,a] is applied to program P the result is:

```
A(P) = m * (a + P)
```

The difference is that in the alternative model, vector arithmetic (i.e., addition and length computation) used for composition and weaving is no longer required. Thus, the alternative composition of *n* aspects is function composition:

$$A_n(A_{n-1}(\ldots(A_1(P))\ldots)) = m_n * (a_n + (m_{n-1} * (a_{n-1} + .. (m_1 * (a_1 + P))..))) \qquad (28)$$

As an example, consider the application of A_1 and A_2 to P, using (28):

$$\begin{aligned} A_2(A_1(P)) &= m_2 * (a_2 + (m_1 * (a_1 + P))) \\ &= m_2 * (\ a_2 + m_1 * a_1 + m_1 * P\ ) \\ &= m_2 * a_2 + m_2 * m_1 * a_1 + m_2 * m_1 * P \end{aligned} \qquad (29)$$

Notice that in this expression, the last two terms ($m_2 * m_1 * a_1$ and $m_2 * m_1 * P$) are identical to those of expression (25), but m_1 disappears from the first term thus eliminating the need to verify that previously applied multipliers do not affect the current development step.

We can verify this in the example of Section 2. Starting with the definitions of (26) and using (28) we have that:

```
Color(Counter(TwoD(Point)))
= setColor + color + counterA + counter +
  counterP*setY + y + counterP*setX + x    (30)
```

where term counterP disappears from the first element of the summation in (27). (See Appendix for details). Thus, the alternative model eliminates the undesirable behavior, from an incremental point of view, of method setColor described in (8).

Does this alternative model have the same power as the AspectJ model? We address this question in the next section.

7 Comparison

All compositions in the AspectJ model can be expressed as compositions in the alternative model. The converse does not hold.

Without loss of generality, consider composition (23) of two aspects and a program in the AspectJ model. The key insight is to refactor both aspects by reifying their multiplicative and additive parts into separate aspects and rearranging the composition expression. For aspects A_1 and A_2 the reification yields four aspects:

$$\begin{array}{lll} Arm_1 = [m_1,0] & \text{and} & Ara_1 = [1,a_1] \\ Arm_2 = [m_2,0] & \text{and} & Ara_2 = [1,a_2] \end{array} \qquad (31)$$

The suffix *rm* stands for reified multiplicative part and *ra* stands for reified additive part. The AspectJ equation in (23) can be expressed as:

$$\begin{aligned} & Arm_2\ (Arm_1\ (Ara_2\ (Ara_1\ (P)))) \\ & \quad = Arm_2\ (Arm_1\ (Ara_2 + Ara_1 + P) \\ & \quad = Arm_2 * Arm_1 * (Ara_2 + Ara_1 + P) \\ & \quad = m_2 * m_1 * (a_2 + a_1 + P) \end{aligned} \qquad (32)$$

The above refactoring and translation of AspectJ compositions to the alternative model is general and can be automated. The appendix illustrates how to obtain the composition expression of AspectJ of the example in Section 2 by using the alternative model.

Unfortunately, the reverse mapping — translating an arbitrary expression of the alternative model composition to one in terms of AspectJ composition — is not possible. The reason is that aspects affect all additive terms of a composition, regardless of when or how they were introduced. In our example, AspectJ sticks the multiplicative term m_1 into the middle of the first term of (25). Certainly the pointcut definition of m_1 could be tweaked so that it does not affect a_2. However, doing that would require a *use-specific* invasive change to the pointcut of m_1. Different composition expressions would require invasive modifications to m_1, as well as other PCAs. This is in contrast to the simple non-invasive refactoring that occurs in the translation of the alternative model, where invasive changes are not needed.

The alternative model also has implications in aspect reuse. Recall the example of Section 2. Consider the following scenario. A new development step requires adding a third dimension, a Z coordinate, to class Point. We can express this step as aspect ThreeD, defined similarly to TwoD (2).

Including all the steps, we want to build three different programs:

- A program that counts execution of setX and setY.
- A program that counts execution of setX, setY, and setZ.
- A program that counts execution of all set methods, including setColor.

Consider the implementation in AspectJ. To cover the three programs, we would need three different versions of aspect Counter. For the first program, we require the constrained version in (9). For the second program, it is necessary to further modify aspect Counter (9) to include execution of setZ. For the third program the original aspect (5) suffices.

On the other hand, the alternative model requires only the original version of aspect Counter. The three programs are expressed as follows:

- ThreeD(Color(Counter(TwoD(Point))))
- Color(Counter(ThreeD(TwoD(Point))))
- Counter(Color(ThreeD(TwoD(Point))))

Bounded quantification allows us to reuse aspects *as-is* (without the need of any invasive changes) and makes aspects behave like traditional modularization technologies in terms of incremental development.

8 Related Work

There exist several tools to help identify, modularize, navigate and understand aspects and the scope their pointcuts. For instance, AJDT [2] provides IDE support in the Eclipse environment for AspectJ. Other IDEs like Emacs, JBuilder, and NetBeans also provide similar functionality. There are specialized tools such as FEAT[9] that uses structural queries to identify and describe concerns. Another tool is the *Concern Manipulation Environment* (CME) [4], whose goal is to provide support for the identification, encapsulation, extraction, and composition of concerns. An overview of concern modelling and supporting tools is in [5].

There has been previous work that addresses the global reasoning characteristic of AspectJ. Kiczales and Mezini [7], propose the use of *Aspect-Aware Interfaces* to support modular reasoning. These interfaces capture the relationship between base code and crosscutting concerns. Aspect-Aware Interfaces describe, among other things, what advice (its type and pointcuts) affect base code. By making this relationship explicit, modular reasoning can be improved.

Careful naming conventions could be imposed to cope with unbounded quantification of AspectJ's composition model. However, doing that contradicts one of the core tenets of AOP, *obliviousness*, which states that programmers should not know about (and thus do not make provisions for) future extensions [5]. Furthermore, regardless of the convention used, the degree of flexibility and reusability is compromised. For instance, if naming conventions were imposed at each developmental stage, reuse would be hindered when aspects were utilized in different stages for which they were originally designed to work on. Part of the problem lies at the shallow abstraction level at which pointcuts work, which describe patterns in syntactic terms. Despite the many alternatives proposed to improve pointcut definition, such as [6], relying only on the pointcut language as opposed to an architectural model is an approach prone to have limitations.

9 Conclusions and Future Work

We showed how AspectJ composition has analogies to vector arithmetic and why it complicates incremental development. Further, we presented an alternative model of composition that treats aspect composition as function composition. The alternative model preserves the full power of AspectJ while eliminating AspectJ's need to verify that previous aspects do not affect the current development step. It can also improve aspect reuse.

We plan to implement this model, using the *Aspect Bench Compiler (ABC)* [1]. The reification of aspect advice as an ITD can be hidden thereby keeping the AspectJ syntax intact. However, to evaluate the full benefit of the alternative model, we will need to extend AspectJ syntax. We believe simple changes can be made to an AspectJ compiler, as mostly what we will be doing is changing the scope of the multiplication operation.

There are several issues yet to be resolved. For instance, aspects can extend other aspects. Thus, it is an open question how can inheritance be integrated into our model. Along the same lines, there are other elements of AspectJ static model that have not been considered in our paper, such as `declare parents` and `declare implements` clauses that can change the relationships between classes and interfaces.

Also, the alternative model should be extended to more clearly address the following issues: aspects with more than one PCA, definition of abstract pointcuts, use of the same pointcut in several advice, different types of join points other than `execution`, other types of advice such as `before` and `around`.

We believe that all the above-mentioned changes will not affect substantially the core results presented in this paper.

Acknowledgements. We thank Oege de Moor, Jeff Gray, Christian Lengauer, Jia Liu, Mark Grechanik, Sahil Thaker, and Wiliam Cook for their comments on drafts of this paper.

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

10 References

[1] Aspect Bench Compiler. http://www.aspectbench.org

[2] Aspect Developement Tools. http://www.eclipse.org/ajdt

[3] AspectJ. Programming Guide. `http://aspectj.org/doc/proguide`

[4] Concern Manipulation Environment (CME) `http://www.eclipse.org/cme/`

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

[6] K. Gybels, J. Brichau. "Arranging Language Features by More Robust Pattern-based Crosscuts". AOSD (2003)

[7] G. Kiczales, M. Mezini. "Aspect-Oriented Programming and Modular Reasoning". International Conference on Software Engineering (ICSE) 2005 (to appear).

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

[9] G. Murphy, A. Lai, R.J. Walker, M.P. Robillard, "Separating Features in Source Code: An Exploratory Study". ICSE (2001).

11 Appendix

Proof of Associativity of +. To prove associativity, we must show that the following holds:

$$(A + B) + C = A + (B + C)$$

Thus we must consider the four cases of concern addition identified in Section 3.1, meaning that A and B have those four combinations. For each of those four cases, we identify the relevant combinations to substitute term C in the above equality. We omit the cases that can be obtained by commuting terms. Let I_i and P_i be ITDs and programs respectively. Two terms match (can be woven together) if the have the same subscripts, otherwise they do not match.

Case 1. $A=P_1$, $B=P_2$. (P_1+P_2) yields programs P_1 and P_2.

- $C=I_1$. Adding I_1 yields I_1-extended class P_1 and class P_2. Conversely, addition of P_2 + I_1 yields the same class and ITD that when added to class P_1 results in the I_1-extended P_1 class and class P_2. Thus the property holds for the this case.

- $C=I_3$. The property holds because the addition of the two sides of the equation yields classes P_1 and P_2, plus the non-matching ITD I_3.

- $C=P_3$. The property holds trivially.

Case 2. $A=I_1$, $B=I_2$. (I_1+I_2) yields ITDs I_1 and I_2.

- $C=P_1$. Adding P_1 yields I_1-extended class P_1 and I_2. Conversely, addition of $I_2 + P_1$ yields P_1 and I_2 that when added to I_1 results in the I_1-extended P_1 class and I_2. Thus the property holds for the this case.

- $C=P_3$. The property holds because addition on both sides yields I_1, I_2, and class P_3.

- $C=I_3$. The property holds trivially in this case.

Case 3. $A=I_1$, $B=P_1$. (I_1+P_1) yields I_1-extended class P_1.

- $C=I_1'$. Adding I_1' yields class P_1 extended with I_1 and I_1'. Conversely, extending P_1 with I_1' and later with I_1 also yields class P_1 extended with I_1 and I_1'.

- $C=I_2$. Adding I_2 yields extended class P_1 and I_2. Conversely, adding P_1 and I_2 yields that class and ITD, when added to I_1 yields I_1-extended class P_1 and I_2.

- $C= P_2$. Similar to previous proof.

Case 4. $A=I_1$, $B=P_2$. (I_1+P_2) yields I_1 and P_2.

- $C=P_1$. Adding P_1, yields extended class P_1 and P_2. Conversely, adding $P_2 + P_1$, yields both classes, and when added I_1 results in extended class P_1 and P_2.

- $C=P_3$. Holds trivially.

- $C=I_2$. Adding I_2, yields extended class P_2 and I_1. Conversely, $P_2 + I_2$ yields extended class P_2 that when added I_1, yields extended class P_2 plus I_1.

Derivation of AspectJ composition of (27)**.** The superscripts indicate the multiplicative (m) or additive (a) part of the aspects of (26).

$$
\begin{aligned}
&|Color \bullet Counter \bullet TwoD \bullet Point| \\
&= [Color^m, Color^a] \bullet [Counter^m, Counter^a] \bullet \\
&\quad [TwoD^m, TwoD^a] \bullet [Point^m, Point^a] \\
&= (Color^m * Counter^m * TwoD^m * Point^m) * \\
&\quad (Color^a + Counter^a + TwoD^a + Point^a) \\
&= (1 * counterP * 1 * 1) * \\
&\quad (Color^a + Counter^a + TwoD^a + Point^a) \\
&= counterP * (Color^a + Counter^a + TwoD^a + Point) \\
&= counterP*Color^a + counterP*Counter^a + \\
&\quad counterP*TwoD^a + counterP*Point^a \\
&= counterP*setColor + counterP*color + \\
&\quad counterP*counterA + counterP*counter + \\
&\quad counterP*setY + counterP*y + \\
&\quad counterP*setX + counterP*x \\
&= counterP*setColor + color + counterA + \\
&\quad counter + counterP*setY + y + \\
&\quad counterP*setX + x \qquad (33)
\end{aligned}
$$

Summation elements such as counterP*counter are reduced to the additive part (ITD or program), counter in this case, because the additive term does not contain join points captured by the PCA.

Derivation of (30)**.** The superscripts indicate the multiplicative (m) or additive (a) part of the aspects of (26)**.**

$$
\begin{aligned}
&Color(Counter(TwoD(Point))) = \\
&\quad Color^m*(Color^a + (Counter^m * (Counter^a + \\
&\qquad (TwoD^m * (TwoD^a + Point))))) \\
&= Color^m * (Color^a + (Counter^m * \\
&\quad (Counter^a + TwoD^m*TwoD^a + TwoD^m*Point) \\
&= Color^m * (Color^a + Counter^m*Counter^a + \\
&\quad Counter^m*TwoD^m*TwoD^a + Couter^m*TwoD^m*Point) \\
&= Color^m * Color^a + Color^m * Counter^m * Counter^a + \\
&\quad Color^m * Counter^m * TwoD^m * TwoD^a \; + \\
&\quad Color^m * Counter^m * TwoD^m * Point \\
&= Color^a + counterP * Counter^a + \\
&\quad counterP * TwoD^a + counterP * Point \\
&= setColor + color + counterP*counterA + \\
&\quad counterP*counter + counterP*setY + \\
&\quad counterP*y + counterP*setX + counterP*x \\
&= setColor + color + counterA + counter + \\
&\quad counterP*setY + y + counterP*setX + x \qquad (34)
\end{aligned}
$$

AspectJ composition of Section 2 using the alternative model. The fist step is to reify the aspects of (26):

$$
\begin{aligned}
&Point^{rm}=[Point^m,0] \qquad Point^{ra}=[1,Point^a] \\
&TwoD^{rm}=[TwoD^m,0] \qquad TwoD^{ra}=[1,TwoD^a] \\
&Counter^{rm}=[Counter^m,0] \quad Counter^{ra}=[1,Counter^a] \\
&Color^{rm} = [Color^m,0] \quad Color^{ra}=[1,Color^a] \qquad (35)
\end{aligned}
$$

Superscript rm stands for reified multiplicative part and ra stands for reified additive part. Thus the composition in the alternative model that achieves the same result as (27) is:

$$
\begin{aligned}
&Color^{rm}(Counter^{rm}(TwoD^{rm}(Point^{rm}(Color^{ra} \\
&\quad (Counter^{ra}(TwoD^{ra}(Point^{ra})))))))) \\
&= Color^{rm}(Counter^{rm}(TwoD^{rm}(Point^{rm}(\\
&\quad Color^{ra} + Counter^{ra} + TwoD^{ra} + Point^{ra})))) \\
&= Counter^{rm} \; (Color^{ra} + Counter^{ra} + TwoD^{ra} + Point^{ra}) \\
&= counterP * (Color^{ra} + Counter^{ra} + TwoD^{ra} + Point^{ra}) \\
&= counterP * Color^a + counterP * Counter^a \; + \\
&\quad counterP * TwoD^a + counterP * Point^a \\
&= counterP*setColor + counterP*color + \\
&\quad counterP*counterA + counterP*counter + \\
&\quad counterP*setY + counterP*y + \\
&\quad counterP*setX + counterP*x \\
&= counterP*setColor + color + counterA + \\
&\quad counter + counterP*setY + y + \\
&\quad counterP*setX + x \qquad (36)
\end{aligned}
$$

The first equality comes from the aspect multiplication identity and (19). The second equality uses aspect multiplication and addition identities and (19). The third equality uses addition and multiplication identity, equation (19), and substitutes $Counter^{rm}$ by counterP. The fourth equality uses distributivity of * over + and substitutes $Point^a$ by Point to yield the same result as (27). The last two equalities expand the multiplicative and additive parts and apply distributivity and identity laws.