Improving Incremental Development with AspectJ using Bounded Quantification

Roberto Lopez-Herrejon & Don Batory
Department of Computer Sciences
University of Texas at Austin

work in progress

Introduction

- Step-wise development with AspectJ is hard
- Illustrate example
- Model of aspect composition using AspectJ
- Present alternative model to support SWD
  - without sacrificing power of AspectJ
  - simplify reasoning about aspects

An Example

of incremental development

class in a graphics application

Incremental Development Example

- Step 1: Point defines 1-dimensional point

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

Step 2: Add Y Coordinate and Method

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

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

`ajc Point.java TwoD.java`

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


Step 3: Count # of Coordinate Changes

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

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

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


Step 4: Add Color Information

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

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

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


Surprise!

- AspectJ produces something different!

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

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

Counter aspect applies to all files in ALL steps!

Paradox

- Building software incrementally:
  - manually
  - automatically using AspectJ
  - can yield different results!
- Redefine Counter could avoid this problem:

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


Well, No...

- It would solve *this* problem, but *not* others
- Ex: if we used the updated Counter, but wanted to build program below, we couldn't do it

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

This code would be missing

The Big Picture

- Premise of **Component-Based Software Engineering (CBSE)** is step-wise development
  - progressively instantiate interfaces of imported components with component implementations
  - reuse components "as is"
- We want to reuse aspect modules as is
  - difficult to do
- Core problem:
  - **need to bound quantification – it is unbounded now**
  - aspect quantification does not distinguish development stages

How We Will Proceed

- Create an algebraic model of how AspectJ composes aspects to discover source of problem
- Present an alternative model of composition that:
  - retains power of AspectJ
  - support incremental development
  - simplifies reasoning with aspects
- Our models reify distinction of inter-type declarations (introductions) and advice

A Model of Introductions

Concern Addition

Model of Inter-Type Declarations

- ITD is a function that maps an input program to an augmented output program

$$Point_2 = TwoD(Point_1)$$

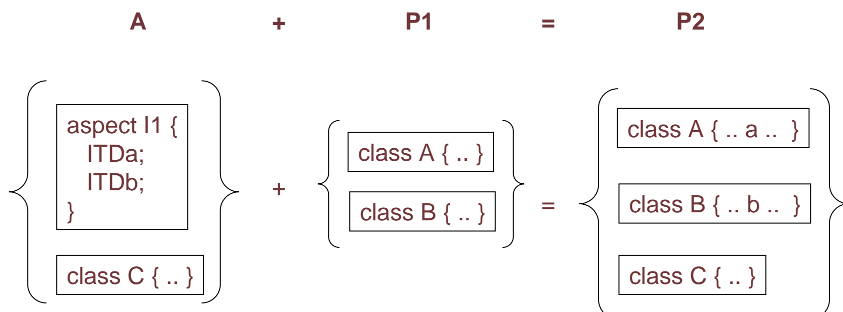
- Appealing to intuition, rewrite above as summation:

$$Point_2 = TwoD + Point_1$$

Concern Addition

Concern Addition

- Adds new data members, methods to existing classes, and can add new classes



Properties of Concern Addition

- **Identity** – denoted by 0
  - 0 is the empty program
  - if X is a program or an aspect file:

$$X = X + 0 = 0 + X$$

- **Commutative** – order in which addition occurs does not matter
  - consistent with AspectJ
- **Associative**: (A + B) + C = A + (B + C)

Properties of Concern Addition

- **Composition**
  - TwoD is a composite ITD

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

→ `TwoD = y + setY`

  - can substitute to produce equivalent defn of $Point_2$

$$\texttt{Point}_2 = \texttt{TwoD} + \texttt{Point}_1$$
$$= \texttt{y} + \texttt{setY} + \texttt{Point}_1$$

A Model of Advice

Concern Multiplication

Advice

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

point cut

advise body

- Advice body (in *red* above) can be regarded as implicit method declaration and call
- We distinguish ideas by
  - making explicit ITD for advice body
  - giving name to each pure advice

Pure Advice – Rewrite Log Aspect

```
aspect Log {
   static void Point.setCalled()
      { System.out.println("set called"); }

   LogP is after(): execution(* Point.set*(..))
      --> Point.setCalled();
}
```

introduction

pure advice

- Not standard AspectJ syntax
- Called **Pure Advice** – separates introduction from advice

Model of Aspects

```
aspect Log {
   static void Point.setCalled()
      { System.out.println("set called");

   LogP is after(): execution(* Point.set*(..))
      --> Point.setCalled();
}
```

- Model as 2-entry vector
  - 1st entry is pure advice (multiplicative part)
  - 2nd entry is ITD (additive part)

```
Log = [ LogP, setCalled ]
```


Another Example

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

   CounterP is after (Point p) :
      execution( * Point.set(..)) && target(p)
      --> Point.IncCntr(p);
}
```

- Modeled by vector:

```
Counter = [ CounterP, counter + IncCntr ]
```


Advice Weaving

- Application of pure advice is operation *

Concern Multiplication

- Let **A** be aspect file with pure advice and **P** be a program
- **A*P** = program resulting from advice in **A** woven into program **P**

Concern Multiplication

- **A2** and **A1** are aspects with pure advice
- **A2 * A1** means apply **A1** first, then **A2**
- Defines precedence ordering of advice

Properties of Concern Multiplication

- **Identity** – denoted by 1
  - 1 is the null advice – pointcut that captures no joinpoints
  - if P is a program and m is a pure advice:

```
P = 1*P
m = 1*m = m*1
```

- **Non-commutative** – order in which multiplication occurs matters
  - well known

Properties of Concern Multiplication

- **Right-Associative**:
  - `m2*m1` means apply `m1` first, then apply `m2`
- **Distributive**: Concern multiplication distributes over concern addition – consequence of quantification

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


Properties of Concern Multiplication

- **Composition**:
  - aspect A1 = [m1, a1]
  - aspect A2 = [m2, a2]
  - ◊ is AspectJ composition operation
- ◊ akin to vector addition:

```
A2 ◊ A1   = [m2, a2] ◊ [m1, a1]
          = [m2*m1, a2+a1]
```


Properties of Concern Multiplication

- Let program P = [1, p]

```
A2 ◊ A1 ◊ P   = [m2, a2] ◊ [m1, a1] ◊ [1,p]
              = [ m2*m1*1, a2+a1+p ]
              = [ m2*m1, a2+a1+p ]
```

- What is the resulting program?

Weaving

- Is "**length**" of vector V
  - follows observable AspectJ semantics

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

- So: $|A2 \Diamond A1 \Diamond P| = m2*m1*(a2+a1+p)$

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

Incremental Development & AspectJ

- Problem seen in expansion

$$|A2 \Diamond A1 \Diamond P| = m2*m1*(a2+a1+p) = m2*m1*a2 + m2*m1*a1 + m2*m1*p$$

- AspectJ programmer needs to know if any of the advice applied in earlier steps affects the code added by the current or later steps

$$m_{j-1}*m_{j-2}*\ldots*m_1*a_j$$

A Fix...

An Alternative Model of Composition

- Treat aspects as functions
- Aspect composition is function composition

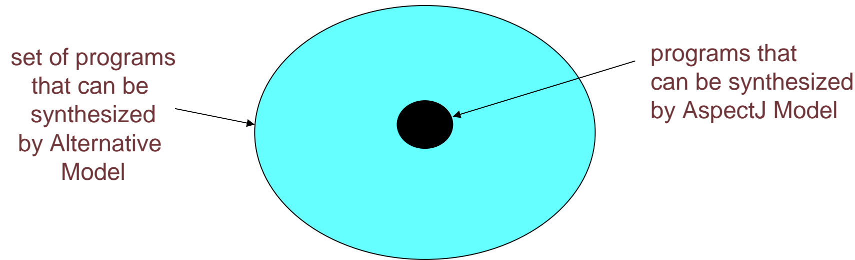
$$A(P) = A \bullet P = m*(a + p)$$

$$A_2 \bullet A_1 \bullet P = m_2*(a_2 + m_1*(a_1 + p)) = m_2*a_2 + m_2*m_1*a_1 + m_2*m_1*p$$

- The terms we don't like (m_1*a_2) are gone!

Comparison of Composition Models

- Alternative Model has more power than AspectJ
  - **provided that aspects are reused as is**



Proof

- Every AspectJ composition can be expressed as an Alternative composition

```
|A2 ◊ A1 ◊ P|   =  m2*m1*( a2+a1+p )

[m2,0] • [m1,0] • [1,a2] • [1,a1] • [1, p]

        = m2*m1*( a2+a1+p )
```


Proof Continued

- Translating arbitrary Alternative Model expression into AspectJ composition is impossible by reusing aspects "as is"
  - can do it if you modify the aspects...

$$A_2 \bullet A_1 \bullet P = m_2 * (a_2 + m_1 * (a_1 + p)) = m_2 * a_2 + m_2 * m_1 * a_1 + m_2 * m_1 * p$$

- Reason: quantification isn't bounded

Implication – Recall Point Example

- Can add 3rd dimension to Point, **ThreeD**
- Can build 3 different programs
  - program that counts executions of setX and setY

    `Color • ThreeD • Counter • TwoD • Point`
  - program that counts execution of setX, setY, setZ

    `Color • Counter • ThreeD • TwoD • Point`
  - program that counts all set methods

    `Counter • Color • ThreeD • TwoD • Point`

Using AspectJ we would need 3 different versions of Counter

Closing Observations

Next Steps

- Features and Aspects are the same thing
  - its their composition models that are different!
  - Feature Oriented Programming and AOP aren't directly comparable
  - instances of a more general model
- Generalize model to handle other issues
  - overriding
  - precedence orders of advice within, among aspect files
- Build prototype of alternative model
  - demonstrate scalability: build large programs using alternative model

Stay Tuned...