modeling crosscutting
in aspect-oriented mechanisms

Hidehiko Masuhara
(University of Tokyo)

joint work with Gregor Kiczales
(University of British Columbia)
aspect-oriented programming

- AOP supports modularization of crosscutting concerns [Kiczales et al. 1997]
- e.g., a drawing editor & a concern: update display when figure moves
- w/o AOP vs. ↓ w/ AOP

(so called) components aspects

Display
update(FigureElement)

Figure
elements
FigureElement
move(int, int)

Point
getX()
getY()
setX(int)
setY(int)

Line
getP1()
getP2()
setP1(Point)
setP2(Point)

DisplayUpdating
update(FigElm)
after(e) : update(FigElm) {
    e.display.update(e)
}
what’s the essence of AOP?

• a naïve model does not capture
  – symmetric mechanism in Hyper/J
  – dynamic mechanism in AspectJ
  – more specialized mechanisms (e.g., Demeter)
  – ...

• we’d like to find a model
  – general enough to capture many mechanisms
  – not too general so that we can see the nature of AOP

crosscutting!
contributions & approach

1. provide a common modeling framework

2. explain modular crosscutting

Aspect SandBox
impl.
impl.
impl.

real AOP languages

AspectJ
Hyper/J
Demeter

[Kiczales’01]
[Ossher&Tarr01]
[Lieberherr’96,97]
talk outline

• implementations of core AOP mechanisms
  – PA: an AspectJ-like (dynamic) mechanism
  – COMPOSITOR: a Hyper/J-like mechanism
  – ( TRAV: a Demeter-like mechanism )
  – ( OC: an AspectJ-like (static) mechanism )

• the modeling framework

• modular crosscutting
  in terms of the modeling framework
PA – pointcuts and advice

- simplified from (dynamic part of) AspectJ [Kiczales01]
- key elements:
  - join point: *point in execution*
i.e., a method call
  - pointcut: specifies *when*
  - advice: specifies *what to do*

\[ \text{after( ):} \]
\[
\text{call(void Point.setX(int))} \\
\text{|| call(void Point.setY(int))} \\
\text{|| call(void Line.setP1(Point))} \\
\text{|| call(void Line.setP2(Point)) } \{ \\
\text{display.update();} \\
\} \\
\]

Modularity
PA: implementation

(def eval-exp
  (lambda (exp env)
    (cond ((call-exp? exp)
            (call-method (call-exp-mname exp)
                         (eval-exp (call-exp-obj exp) env)
                         (eval-rands (call-exp-rands exp) env)))
       ...
       )))

(define-struct call-jp (mname target args))

(define call-method
  (lambda (mname obj args)
    (let*((jp (make-call-jp mname obj args))
          (method (lookup-method jp))
          (advice (lookup-advice jp)))
      (execute-advice advice jp
                      (lambda ()
                        (execute-method method jp))))))

an interpreter (à la EOPL)

a join point represents a method call

a method call is to:

- create a join point
- identify a method
- identify advice decls.
- execute advice decls.
- execute method
observations from PA implementation

- method and advice are treated similarly: lookup & execute
  ➔ symmetric model

- join points come from execution
  ➔ “weaving into components” is not good
  ➔ weaving in the third space (i.e., execution)
COMPOSITOR –
class composition

simplified from Hyper/J [Ossher01]

• class hierarchy for each concern (no dominant modularity)
• composition of class hierarchies to get an executable
• composition specification

```java
Observable
moved() {
    display.update(this);
}
```

```java
Display
update(FigureElement)
```

```java
Figure
Display.update(FigureElement)
```

```java
FigureElement
Display.update(FigureElement)
```

```java
Observable
moved() {
    display.update(this);
}
```

```java
Point
getX()
getY()
setX(int)
setY(int)
moveBy(int,int)
```

```java
Line
getP1()
getP2()
setP1(Point)
setP2(Point)
moveBy(int,int)
```

match Point.setX with Observable.moved
match Point.setY with Observable.moved
match Line.setP1 with Observable.moved
match Line.setP2 with Observable.moved
```
COMPOSITOR: implementation

source-to-source translation
1. computes all possible combinations *
2. determines whether each should be merged
3. merges bodies & adds to program

(* very naïve approach; just for explanation)
(define compositor:weave
  (lambda (pgm-a pgm-b relationships)
    (let loop ((pgm (make-program '()))
               (seeds (compute-seeds pgm-a pgm-b)))
      (if (not (null? seeds))
          (let ((signature (all-match (car seeds) relationships)))
            (if signature
                (let* ((jp (car seeds))
                        (decl (merge-decls jp relationships)))
                  (loop (add-decl-to-pgm decl pgm signature)
                        (remove-subsets jp (cdr seeds)))))
                (loop pgm (cdr seeds)))))))

generate combinations of methods

test all methods have matching signatures

merge method bodies and install
observations from COMPOSITOR implementation

- no dominant modularity
  ➔ symmetric model
- join points are not only from pgm-a (nor pgm-b)
  ➔ “weaving into components” is not good
  ➔ weaving in the third space
- matching rule can be modified
  ➔ weaving parameters
similarly implemented

- **TRAV**: Demeter/DemeterJ/DJ [Liberrherr97], etc.
  - traversals through object graphs
    - modular specification: “where to go” & “what to do” otherwise scattered over classes

- **OC**: AspectJ’s introductions or ITD [Kiczales01]
  (also in Flavors, etc. [Cannon82]...)
  - can declare methods/fields outside of the class declarations
the modeling framework: PA’s case

method & advice are parallel

method & advice are parallel

weaving happens at method calls in computation
modeling framework: COMPOSITOR’s case

A - methods & fields

ID - signature matching

EFF - provide decls

B - methods & fields

IDB - signature matching

EFFB - provide decls

META - composition rules

(compositor:weave <program-a> <program-b>
*match Point.setX with Observable.moved
match Point.setY with Observable.moved
match Line.setP1 with Observable.moved
match Line.setP2 with Observable.moved*)

XJP - set of decls

X - composed programs
the modeling framework

A - program

B - program

X - computation or program

A&B are parallel

weaving happens at X

EFF\textsubscript{B} - means of effecting

EFF\textsubscript{A} - means of effecting

ID\textsubscript{A} - means of identifying

ID\textsubscript{B} - means of identifying

X_{JP} - join point
## models for 4 mechanisms

<table>
<thead>
<tr>
<th></th>
<th>PA</th>
<th>TRAV</th>
<th>COMPOSITOR</th>
<th>OC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>X</strong></td>
<td>program execution</td>
<td>traversal execution</td>
<td>composed program</td>
<td>combined program</td>
</tr>
<tr>
<td><strong>XJP</strong></td>
<td>method calls</td>
<td>arrival at each object</td>
<td>declarations in X</td>
<td>class declarations</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>c, m, f declarations</td>
<td>c, f declarations</td>
<td>c, m, f declarations</td>
<td>c declarations w/o OC declarations</td>
</tr>
<tr>
<td><strong>AID</strong></td>
<td>m signatures, etc.</td>
<td>c, f signatures</td>
<td>c, m, f signatures</td>
<td>method signatures</td>
</tr>
<tr>
<td><strong>A_{eff}</strong></td>
<td>execute method body</td>
<td>provide reachability</td>
<td>provide declarations</td>
<td>provide declarations</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>advice declarations</td>
<td>traversal spec. &amp; visitor</td>
<td>(= A)</td>
<td>OC method declarations</td>
</tr>
<tr>
<td><strong>B_{ID}</strong></td>
<td>pointcuts</td>
<td>traversal spec.</td>
<td>(= A_{ID})</td>
<td>effective method signatures</td>
</tr>
<tr>
<td><strong>B_{eff}</strong></td>
<td>execute advice body</td>
<td>call visitor &amp; continue</td>
<td>(= A_{eff})</td>
<td>copy method declarations</td>
</tr>
<tr>
<td><strong>META</strong></td>
<td>none</td>
<td>none</td>
<td>match &amp; merge rules</td>
<td>none</td>
</tr>
</tbody>
</table>
what’s modular crosscutting?

• it is said: “AOP supports modular crosscutting” but what is it?

• the modeling framework can explain: two modules in A&B crosscut when projections of the modules into X intersect and neither is subset of the other
an example of modular crosscutting in PA

"Line and DisplayUpdating crosscut in the execution"
what’s modular crosscutting?

two modules in A&B crosscut when

projections of the modules

into X intersect and

neither is subset of the other
examples of modular crosscutting

PA

call to l.setP1
call to l.getP1

call to p.setX
call to p.getX

TRAV

arrive at l
arrive at p
arrive at p'

arrive at l
arrive at p

COMPOSITOR

Observable moved

OC

DisplayMethods

Point.draw
Line.draw
related work

• comparison *two* AOP mechanisms; e.g., Demeter vs. AspectJ [Lieberherr97]

• formal models for *particular* AOP mechanism [Wand+01], [Lämmel01], etc.

• Filman-Friedman’s claim on non-invasiveness, or “quantified assertions over programs written by oblivious programmers”
  – not explicit in our framework; suggesting invasive AOP mechanisms is possible
summary

• 3 part modeling framework
  – elements from A&B meet at JP in X
  – based on executable implementations
    www.cs.ubc.ca/labs/spl/projects/asb.html
• explanation of modular crosscutting
  – in terms of projections of A&B modules into X
• future work:
  – discuss more features in AOP on the framework
    e.g., non-invasiveness, remodularization, …
  – unified implementation and formalization
  – apply to foundational work: semantics [Wand01,02],
    compilation [Masuhara02,03], new feature designs…
TRAV – traversals

- based on Demeter/DemeterJ/DJ [Liberrherr97]
- traversals through object graphs
  - specification: “where to go” & “what to do”
  - otherwise scattered among classes
- e.g., counting FigureElements in a Figure

```java
Visitor counter = new CountElementsVisitor();
traverse("from Figure to FigureElement", fig, counter);
```
implementation of TRAV

- semantics: visit all objects that can lead to goals
- naïve implementation: a traversal engine
  - walks over object graph
  - locates current object in the spec.
  - visits & continues walking if reachable in terms of class graph
implementation of TRAV

• weaver = traversal engine

(define trav:weave
  (lambda (trav-spec root visitor)
    (let arrive ((obj root)
                 (path (make-path (object-cname root))))
      (call-visitor visitor obj)
      (for-each (lambda (fname)
                 (let* ((next-obj (get-field fname obj))
                        (next-cname (object-cname next-obj))
                        (next-path (extend-path path
                                     next-cname)))
                    (if (match? next-path trav-spec)
                        (arrive next-obj next-path))))
      (object->fnames obj))))

• visit object
• match path vs. spec
• and cont.
model for TRAV

- class & field decls.
- provide reachability
- traversal spec.
- traversal spec. & visitor desc.
- visit & continue
- arrival at object

Diagram:
- Class and field declarations
- Provide reachability
- Traversal specifications
- Traversal specifications and visitor description
- Arrival at object
OC – open classes

• based on AspectJ’s introductions [Kiczales01] Flavors, etc. [Cannon82]...

• can declare methods/fields outside of the class declarations

• example: add drawing functionality

```java
class DisplayMethods {
    void Point.draw() { Graphics.drawOval(...); }
    void Line.draw() { Graphics.drawLine(...); }
}
```
implementation of OC

• a special case of COMPOSITOR
  – a source-to-source translator
  – class decls \times oc decls → program

(define oc:weave
  (lambda (pgm);→ pgm
    (let ((pgm (remove-oc-mdecls pgm))
          (oc-mdecls (gather-oc-mdecls pgm)))
      (make-pgm
       (map (lambda (cdecl)
              (let* ((cname (class-decl-cname cdecl))
                     (sname (class-decl-sname cdecl))
                     (per-class-oc-mdecls (lookup-oc-mdecls cname oc-mdecls)))
                   (make-class-decl cname sname
                                    (append (class-decl-decls cdecl)
                                            (copy-oc-mdecls cname per-class-oc-mdecls))))
            (pgm-class-decls pgm))))))

• A
• B
• a new program in X
• cdecl is a jp

• \(\text{ID}_B\)
• \(\text{EFF}_A\) and \(\text{EFF}_B\)
model for OC

A - OO decls.

EFF A - provide decls.

ID A - signature matching

EFF B - copy decls.

ID B - signature matching

B - OC decls.

JP - class

X - composed programs
QB – query-based browser

- a customizable code exploration tool [Rajagopolan02]
- takes parameters:
  - properties to extract
  - order of properties
- can give different views of a program; e.g., group classes by method names
QB – query-based browser

- Classes with defined methods
- Method names with defining classes
implementation of QB

1. extract metaobjects
2. build envs.
3. test query against each env
4. add nodes to tree guided by the var. list
implementation of QB

(define qb:unweave
  (lambda (pgm query tree-vars);->tree
    (let* ((metaobjects (elaborate-program pgm))
           (all-envs (possible-envs (query-vars query)
                                   metaobjects))
           (tree (make-empty-tree)))
      (for-each (lambda (env)
          (if (match? env query)
              (let ((vals (map (lambda (var)
                                (lookup-var var env))
                               tree-vars)))
                (add-to-tree! vals tree)))))
    all-envs)
  tree)))

• A to isomorphic X
• tuples of jps
• B
• ID_B
• EFF_B
• return B
model for QB

A: programs

JP: class, method, ...

X: programs

META: variable list

B: tree

this is *unweaving* process