Modeling Crosscutting in Aspect-Oriented Mechanisms

Hidehiko Masuhara[1] and Gregor Kiczales[2]

[1]University of Tokyo
masuhara@acm.org
[2]University of British Columbia
gregor@cs.ubc.ca

Abstract. Modeling four aspect-oriented programming mechanisms shows the way in which each supports modular crosscutting. Comparing the models produces a clear three part characterization of what is required to support crosscutting structure: a common frame of reference that two (or more) programs can use to connect with each other and each provide their semantic contribution.

1 Introduction

A number of tools have been proposed for aspect-oriented (AO) software development. But to date there is no generally agreed upon definition of what makes a technique aspect-oriented. This paper takes a step in that direction by presenting a framework that can be used to explain how four proposed mechanisms support modular crosscutting:

– Pointcuts and advice as in AspectJ [10, 11].
– Traversal specifications as in Demeter[15], DemeterJ[16] and DJ[14, 21].
– Class composition as in Hyper/J[22, 23].
– Open classes as in AspectJ.[1]

We capture the core semantics of these mechanisms by modeling the weaving process by which they are implemented. The models define a weaving process as taking two programs and coordinating their coming together into a single combined computation. In the case of mechanisms like Hyper/J, where the semantics is defined in terms of code-processing, the model describes combining input programs to produce a single combined program.

A critical property of the models is that they describe the join points as existing in the result of the weaving process rather than being in either of the input programs. This yields a three-part description of crosscutting structure – in terms of how the two input programs *crosscut each other with respect to* the result computation or program. This three-part description is essential to model all four mechanisms, and to model an arbitrary number of crosscutting programs.

[1] This functionality originated in Flavors [19, 28] where Flavor declarations did not lexically contain their methods and is found in several other languages; the term open class is due to [5, 18]. AspectJ used to call this feature introduction, and now calls it inter-type declarations.

The paper is organized as follows:

Section 2 presents a simple object-oriented (OO) language that is used as a basis for discussion throughout the paper. This includes an example program in the language, and key elements of a simple interpreter for the language.

Section 3 presents the common structure of all the models.

Section 4 has four parts, each of which shows how one of the four mechanisms is modeled. The discussion in each part first outlines the core semantics to be addressed, by showing how that semantics could be embedded in the language of Section 2. This starts with an example program and an implementation of the semantics, as a modification to the implementation of Section 2.2. We then show the model and use the implementation to intuitively validate the model.

Section 5 shows that the simplifications of Section 4 do not invalidate the models of the real systems. This is done by showing that a number of features originally left out of the models in Section 4 can easily be added to the models.

Section 6 presents the model of crosscutting structure and uses it to describe the crosscutting in the examples from Section 4.

Section 7 discusses related work, Section 8 presents future work, and Section 9 is a summary.

The paper assumes prior knowledge of all four AO mechanisms and a reading familiarity with simple Scheme interpreters of OO languages. But, at least on a first pass through, the paper can be read in subsets. Readers unfamiliar with all the mechanisms can skip the corresponding sub-sections of Section 4. Readers can avoid much of the Scheme code by skipping Sections 2.2 and 4.[1-4].1; the effect of this will be to read the models, without the validation of their appropriateness.

The code in this paper is available online.[2] This includes all the model implementations, as well as the example programs written in both the full systems (AspectJ, Hyper/J, etc.) as well as our core models of those systems.

2 BASE – An Object-Oriented Language

This section describes a simple object-oriented language, called BASE, which is the basis for the rest of the paper. The description has three parts: a simple intuitive description of the language, a sample program in the language, and a simple Scheme interpreter for the language. The discussion of each of the four mechanisms in Section 4 follows this same structure, and builds on both the example program and the model interpreter.

[2] See http://www.cs.ubc.ca/labs/spl/projects/asb.html.

BASE can be seen as a core subset of Java. It is a single-inheritance OO language, without interfaces, overloading or multi-methods. Programs in BASE are written in a Java-like syntax.[3]

2.1 Sample Program

A simple program written in the BASE language is:

```
class Figure {
  List elements = new LinkedList();
}

class FigureElement {
  Display display;
}

class Point extends FigureElement {
  int x;
  int y;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) { this.x = x; }
  void setY(int y) { this.y = y; }
}

class Line extends FigureElement {
  Point p1;
  Point p2;

  int getP1() { return p1; }
  int getP2() { return p2; }

  void setP1(Point p1) { this.p1 = p1; }
  void setP2(Point p2) { this.p2 = p2; }
}
```

This code implements a simple figures package. A `Figure` is comprised of a collection of `FigureElements`. There are two kinds of `FigureElements`, `Point` and `Line`. Both points and lines have the usual structure, and getter and setter methods. See [11] for a more detailed discussion of this example.

[3] In our actual implementation, BASE has a more Scheme-like syntax. We use a Java-like syntax in the paper to help the reader distinguish between code written in BASE and the Scheme code that implements BASE. As part of using Java syntax, we show code with a return statement, even though the actual BASE language has only expressions, no statements.

2.2 Model Implementation

This section presents the model implementation of BASE. The implementation is an interpreter structured in a manner similar to Chapter 5 of [8]. We show the main structure of the interpreter, including key data structures and procedures. A number of minor helper procedures are not included in this presentation.

For simplicity our model implementation does not include a static type checker, we simply assume the program checks. (We can do this because the language has no overloading, or other features that cause static typing to have semantic effect on correct code.)

The interpreter is written in Scheme [9]. We use a style in which variable names represent the type of their value. (Not actual Hungarian notation.) To save space, some of these names are abbreviated. The following table shows the naming and abbreviation conventions used in the code.

abbreviation	*type of value*
`pgm`	program
`cname, mname…`	class, method, field and super name
`decl`	declaration
`cdecl, mdecl…`	class, method and field declaration
`param`	parameter
`id`	identifier
`methods`	list of methods
`fields`	list of fields
`cdecls, mdecls…`	list of class, method… declarations
…	…
`env`	environment
`exp`	expression

The main data structures of the interpreter include AST structures that represent class, method and field declarations, and expressions that represent code.

```
(define-struct class-decl  (cname sname decls))
(define-struct method-decl (rtype mname params body))

(define-struct exp ())
(define-struct (if-exp exp) (test then else))
<and many sub-types of exp>
```

There are class and method structures that represent total effective class and method definitions. While a class declaration includes method declarations for only those methods defined lexically in the class, a class structure includes method structures for methods defined in the class as well as inherited methods.

```
(define-struct class  (cname sname fields methods))
(define-struct method (cname mname params body))
(define-struct field  (type fname))
```

There are also environment data structures that hold variable values and class definitions. The code for those is not shown here.

The entry point of the interpreter is the `eval-program` procedure. It first walks over the declarations to produce the class and method structures; the global variable `*classes*` is bound to a list of the class structures. Then `eval-expression` is called with an expression that calls the main entry point.[4]

```
(define eval-program
 (lambda (pgm)
   (set! *classes* (init-classes))
   (elaborate-class-decls! (program-decls pgm))
   (eval-exp (parse-exp "(new Main()).main()")
             (make-empty-env))))
```

The `eval-expression` procedure is a standard recursive evaluator. Within the structure of `eval-expression`, we identify specific helper procedures, `call-method`, `get-field` and `set-field`, for evaluating method calls and field accesses. Separating these helper procedures, rather than inlining them in `eval-expression` is done to simplify the presentation in later sections of the paper.

```
(define eval-expression
  (lambda (exp env)
    (cond
      ((literal-exp? exp) (literal-exp-datum exp))
      ((if-exp?      exp) ...)
      ...
      ((method-call-exp? exp)
       (let ((mname   (method-call-exp-mname exp))
             (obj-exp (method-call-exp-obj-exp exp))
             (rands   (method-call-exp-rands exp))
         (call-method mname
                      (eval-exp obj-exp env)
                      (eval-rands rands env))))
      ((get-field-exp? exp)
       (let ((obj (apply-env env 'this))
             (fname (get-field-exp-fname exp)))
         (get-field obj fname)))
      ((set-field-exp? exp)
        ...
         (set-field obj fname val))))))

(define call-method
  (lambda (mname obj args)
    (let ((method (lookup-method mname obj)))
      (execute-method method obj args))))

(define get-field
  (lambda (obj fname)
    <get the field value from the object>))
```

[4] In the model implementation on the website, this entry point is the main method of the class called `Main`. To save space, the example code in Section 2.1 doesn't include a `Main` class.

```
(define set-field
  (lambda (obj fname val)
    <set the field value in the object>))
```

The `lookup-method` procedure simply finds the method in the methods of the class, if no matching method is found an error is signaled.

```
(define lookup-method (mname obj)
  (let* ((cname   (object-cname obj))
         (methods (cname->methods cname)))
    <scan methods looking for one matching mname>
    ))
```

The `execute-method` procedure simply evaluates the body of the method in an environment with `this` bound to the object, and the method parameter identifiers bound to the arguments.

```
(define execute-method (method obj args)
  (eval-expression (method-body method)
    (make-env (cons 'this (method-ids method))
              (cons obj args))))
```

Together with a modest number of helper procedures not shown here, this provides a complete implementation of the BASE language.

3 The Modeling Framework

This short section presents the modeling framework in general terms. (Readers who prefer bottom-up presentation of general concepts may want to skip directly to section 4.)

The framework models each AO mechanism as a *weaver* that combines two input programs to produce either a program or a computation. Each weaver is modeled as an 11-tuple:

$$\langle X, X_{JP}, A, A_{ID}, A_{EFF}, A_{MOD}, B, B_{ID}, B_{EFF}, B_{MOD}, META\rangle$$

A and B are the languages in which the input programs p_A and p_B are written. X is the result domain of the weaving process, which is usually a computation, but can be a third language to model systems like Hyper/J, where the semantics is defined in terms of manipulating code.

X_{JP} is the join points in X.

A_{ID} and B_{ID} are the means, in the languages A and B, of identifying elements of X_{JP} (the join points in X).

A_{EFF} and B_{EFF} are the means, in the languages A and B, of effecting semantics at identified join points.

A_{MOD} and B_{MOD} are the units of modularity in the languages A and B. All discussion of the units of modularity is deferred to Section 6.

META is an optional meta-language for parameterizing the weaving process. In cases where META is not used we simply leave it out of the model.

A *weaving process* is defined as a procedure with signature:

Table 1. Summary of four models. In this table the single letters 'c', 'm' and 'f' are abbreviations for class, method and field respectively.

	PA	TRAV	COMPOSITOR	OC
X	program execution	traversal execution	composed program	combined program
X_{JP}	method calls	arrival at each object	declarations in X	class declarations
A	c, m, f declarations	c, f declarations	c, m, f declarations	c declarations w/o OC declarations
A_{ID}	m signatures, etc.	c, f signatures	c, m, f signatures	method signatures
A_{EFF}	execute method body	provide reachability	provide declarations	provide declarations
B	advice declarations	traversal spec. & visitor	(= A)	OC method declarations
B_{ID}	pointcuts	traversal spec.	(= A_{ID})	effective method signatures
B_{EFF}	execute advice body	call visitor & continue	(= A_{EFF})	copy method declarations
META	*none*	*none*	match & merge rules	*none*

$$A \times B \times \text{META} \rightarrow X$$

That is, it accepts three programs, p_A, p_B and p_{META}, written in the A, B and META languages, and produces either a computation or a new program.

A critical property of this model is that it models A, B and X as distinct entities, and models weaving as combining the semantics of programs in A and B at join points in X. This differs from models that have two elements (i.e. A and B), and characterize B as merging with A at join points in A. The implications of this three-part model are discussed further in Section 6, but two points are worth discussing here.

In general, A and B can be different. This means that A_{ID} and B_{ID} can differ, as can A_{EFF} and B_{EFF}. For example, in the model of AspectJ presented in Section 4.1, method declarations are modeled as elements of the A language, and advice are modeled as elements of B. Both method declarations and advice can affect what happens at a method call. But their means of identifying the call is different (a method signature in a method declaration vs. a pointcut in an advice), and their means of effecting semantics of the call is different (execution of a method differs from execution of after advice).

In cases where A and B are different, it is often the case that one of them can be seen as more similar in structure to X than the other. Again, in the AspectJ case, X will be the execution of the objects, which can be seen as more similar in structure to A (the classes and methods) than to B (the advice). In such cases, we will always use A as the name of the one that is more similar to X. But it will be critical to remember that A is not the same as X, it is just highly similar in structure to X. A key property of this framework is to distinguish A and X even when they are quite similar.

4 Four Mechanisms in Terms of the Model

This section presents models of four mechanisms found in current AO systems. Table 1 shows a summary of the four models. For each mechanism, we first present a simplified, or core version of its semantics, by providing an intuitive description and a short example program. These are based on the corresponding material developed for the BASE semantics in Section 2. We then present the model, which is derived from the framework by filling in the eleven parameters. Each sub-section ends with an intuitive validation of the model, which is done by showing how elements of the implementation correspond to elements of the model.

4.1 PA – Pointcuts and Advice

This section shows how the pointcut and advice mechanism in AspectJ can be modeled in terms of the modeling framework.

We first present a simplified, or core, version of the pointcut and advice mechanism semantics; we call this core semantics PA. As compared to AspectJ, PA has only method call join points, after advice declarations, and call, target, && and || pointcuts. While this leaves out significant AspectJ functionality, it suffices to capture the important elements, and later in the paper Section 5.1 shows that the missing functionality can be added without requiring changes to the model.

As an example of the PA functionality, consider the following after advice, which implements display updating functionality similar to that in [11].

```
after(FigureElement fe):
      (call(void Point.setX(int))
        || call(void Point.setY(int))
        || call(void Line.setP1(Point))
        || call(void Line.setP2(Point)))
      && target(fe) {
   fe.display.update(fe);
}
```
[5]

Implementation of PA. This section presents an interpreter for PA, based on a few small changes and additions to the interpreter for BASE.

First, we define a structure used to represent dynamic join points:

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

[5] In full AspectJ, this could be written as:

```
after(FigureElement fe):
     call(void FigureElement+.set*(..)) && target(fe) {
   fe.display.update(fe);
}
```

But we write the longer form because we do not implement the required type pattern and wildcarding functionality in this paper.

This structure says that the dynamic values at a join point – remember that for now PA only has method call join points – include the name of the method being called, the object that is the target of the call, and a list of the arguments to the call.

The weave procedure is defined as:

```
(define pa:weave
  (lambda (pgm); -> computation
    (fluid-let ((pgm
                  (remove-advice-decls pgm))
                (*advice-decls*
                  (gather-advice-decls pgm)))
      (eval-program pgm))))
```

The weaver first separates the advice declarations from the rest of the program, leaving it with an ordinary BASE program, as well as a list of advice declarations. The weaver then proceeds to evaluate the BASE program.

The `call-method` procedure is modified to create method call join point structures, and check whether any advice declarations match the join point; these advice declarations are run after executing the method itself.

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

In addition to redefining `call-method`, we also redefine `lookup-` and `execute-method` to take a single jp structure as their argument, rather than taking `mname`, `obj` and `args` separately. This simple change is not shown here.

The role of `lookup-advice` is to take a jp structure and look in `*advice-decls*` to find which advice declarations have a pointcut that matches the join point. The result of `lookup-advice` is a list of `adv-match` structures. Each such structure represents the fact that a particular advice declaration matched, and includes bindings of parameters of the advice to values in the context of the join point (i.e. `fe` is bound to the figure element).

```
(define-struct adv-match (adv-decl ptc-match))
(define-struct ptc-match (ids vals))
```

`lookup-advice` works simply by looping through all the advice declarations, calling `pointcut-matches` to see if each advice declaration's pointcut matches the join point.

```
 (define lookup-advice
  (lambda (jp)
    (remove*[6] #f
      (map (lambda (adecl)
             (let* ((ptc (advice-decl-ptc adecl))
```

[6] Remove all occurrences of an item from a list.

```
                    (ptc-match
                      (pointcut-matches ptc jp)))
               (if (not ptc-match)
                   #f
                   (make-adv-match adecl
                                   ptc-match))))
         *advice-decls*))))
```

`pointcut-matches` is simply a case-based test to see whether a given pointcut matches the join point. If not, it returns false, otherwise it returns a `ptc-match` structure. Note that `target` is currently the only pointcut that binds parameters.

```
(define pointcut-matches
  (lambda (ptc jp)
    (cond ((call-pointcut? ptc)
           (and (eq? (call-pointcut-mname ptc)
                     (call-jp-mname jp))
                (make-ptc-match '() '())))
          ((target-pointcut? ptc)
           (make-ptc-match
             (list (target-pointcut-id ptc))
             (list (call-jp-target jp))))
          ((and-pointcut? ptc) ...)
          ((or-pointcut?  ptc) ...)
          )))
```

The `execute-advice` procedure takes a list of advice match structures, a jp structure and a thunk as arguments. The thunk implements the computation at the join point independent of any advice. As shown in the `call-method` procedure above, at method call join points, the thunk implements the behavior of `call-method` in the original BASE system. Note that `execute-advice` must be able to handle a list of matching advice, because join points can have more than one matching advice.

```
(define execute-advice
  (lambda (adv-matches jp thunk)
    (let ((result (thunk)))
      (for-each (lambda (adv-match)
                  (execute-one-advice adv-match
                                      jp))
                adv-matches)
      result)))
```

Model of PA. The model of PA in terms of the framework is as follows. Note that we use italics to identify parts of PA semantics that are deferred to Section 5.

X	Execution of combined programs
X_{JP}	method calls, *and field gets and sets*
A	Class, method *and field* declarations
A_{ID}	method *and field* signatures
A_{EFF}	execute method body, *get and set field value*
B	advice declarations with pointcuts

B_{ID}	Pointcuts
B_{EFF}	execute advice body *before*, after *and around* method

We use the implementation as intuitive evidence that the model is realizable and appropriate. We do this by matching the model parameters to corresponding elements in the implementation code.

A and B are clearly modeled in the implementation of `pa:weave`. A program p_A in the language A is the class declarations from the complete program; a program p_B in the language B is the advice declarations, with their associated pointcuts. X is the complete computation, which `pa:weave` produces by calling `eval-program`. In this case A plays a primary role over B, as the weaver proceeds by running A, calling advice from B when appropriate.

The revised implementation of `call-method` models method call join points (X_{JP}) as the points in the flow of control when a method is called. The jp structure models the kind of join point, as well as the values available in the context of the join point.[7]

In A, the complete signatures of method declarations are the means of identifying join points (A_{ID}), and execution of the method body is the means of specifying semantics at the join points (A_{EFF}). Taken together, these say, "when execution reaches a call to an object of this class with this method name, then execute this code."

In B, the means of identifying join points (B_{ID}) is pointcuts, and is modeled by `pointcut-matches`. The means of effecting semantics (B_{EFF}) is execution of the advice body after continuing with the join point, and is modeled by `execute-advice`.

4.2 TRAV – Traversals

The Demeter systems (Demeter, DemeterJ and DJ) provide a mechanism that enables programmers to implement traversals through object graphs in a succinct declarative fashion. The effect of this functionality is to allow the programmer to define, in a modular way, a traversal that would otherwise require code scattered among a number of classes. They work by defining the traversal as well as what actions to take at points along the traversal.

In this section, we work with a simple traversal semantics, called TRAV. TRAV supports declarative description of the traversal, but not whether to call the visitor at each object in the traversal; the visitor is simply called at every object in the traversal. As with PA above, while this omits important functionality, that omission does not impact the suitability of the general framework. Section 5 shows how the omitted functionality can be added.

An example of a program fragment written using TRAV is:

[7] The semantics of values in the context of a join point and how the `this`, `target` and `args` pointcuts access those values is more complex than this in AspectJ, because proceed can change the values that `args` sees. Doing it properly makes the code more complex, but does not impact the modeling framework or the model of PA.

```
Visitor counter = new CountElementsVisitor();

traverse("from Figure to FigureElement",
         fig,
         counter);
```

This code fragment implements the behavior of visiting all the `FigureElements` reachable from a `Figure`. The first argument to `traverse` is called a traversal specification; it describes the path to follow to each object to be visited. The second argument is a root object, where the traversal starts. The third argument is a visitor, which defines behavior at each traversed object. In this case the traversal mechanism is taking care of iterating through the elements of the figure, and following down through line objects to reach point objects. The visitor is called on every object in the traversal, including the `Figure` as well as the `List` that holds the `FigureElements`. The actual visitor must decide which objects to count, i.e. not to count the `List`.

We preserve the critical property of Demeter that the range of traversal is based on reachability information from the class graph, in addition to information about the dynamic class of the current object. Therefore, when the traversal comes to a `Line` object, for example, it goes on to the `Point` objects referenced by the `Line`; but it does not go on to the `Display` object, because the traversal specification says it is looking for `FigureElement` and the class graph shows there are no ways to reach a `FigureElement` from a `Display`.

Implementation of TRAV. The weaver implementation for TRAV is the procedure `trav:weave`. We modify `eval-expression` from the BASE interpreter to call `trav:weave` to implement the new `traverse` primitive. The definition of `trav:weave` is:

```
(define trav:weave
  (lambda (trav-spec root visitor)
    (let arrive ((obj  root) ;arrival at obj is a jp
                 (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)))))
```

The traversal process can be seen as a simple depth-first walk with a navigator that restricts the walk. When the walk arrives at an object, it calls the visitor with the object, and then recursively walks the objects referenced by the object. At each step along the way, it first checks with the navigator about whether or not to proceed.

The navigator checks whether to traverse to an object in two steps. It first locates possible positions of the object in the traversal specification. The traversal specifications following those positions are then tested against the class graph. The test succeeds when there exists a path on the class graph that matches a remaining specification.

We implement this in a simple way. We assume the root object is always a legal root of the traversal specification. In order to locate positions in the specification, the code manages a path that keeps track of a sequence of classes walked from the root. The `match?` procedure checks whether the whole path matches the traversal specification by using two sub-procedures that correspond to the two steps above.

```
(define match?
  (lambda (path trav-spec)
    (let ((residual-spec (match-path path trav-spec)))
      (match-class-graph?
        residual-spec (path-last-cname path)))))
```

The `match-path` procedure matches the path to the traversal specification by repeatedly matching each class in the path from the root.[8] The matching algorithm is implemented by simple conditional cases on the kind of the directive at the head of the specification. The code returns either a remaining specification for partially matched cases, an empty specification for completely matched cases, or false for unmatched cases. When a class matches more than one position in the specification, an `or`-specification is returned as a result.

```
(define match-path
  (lambda (path spec);->spec
    (let loop ((cnames (path->cnames path))
               (spec spec))
      (if (null? cnames) spec
          (loop (cdr cnames)
                (match-cname (car cnames) spec))))))
(define match-cname
  (lambda (cname spec);->spec or #f
    (cond ((null? spec) #f) ; unmatched
          ((to-spec? spec)
           (if (subclass? cname (to-spec-cname spec))
               (make-or-spec '()
                 (list spec (spec-next spec)))
               spec))
          ...)))
```

The `match-class-graph?` procedure matches the remaining specification against the class graph, and returns true or false. For a `to`-specification, the matching is simply subsumed by the reachability to the specified class. Note that the reachability is decided by checking the class graph. In the implementation, the procedure `reachable?`, not shown here, does this by using global variable `*classes*`.

```
(define match-class-graph?
  (lambda (spec root-cname);->boolean
    (let loop ((spec spec)
```

[8] A more sophisticated implementation would implement these steps with a state transition machine so that it could avoid complicated checks at each object.

```
                (cname root-cname))
        (cond ((eq? spec #f) #f) ; already unmatched
              ((to-spec? spec)
               (reachable? cname (to-spec-cname spec)))
              ...))))
```

Calling the visitor involves calling the visit method with the object as its argument.

```
(define call-visitor
  (lambda (visitor obj)
    (call-method 'visit visitor (list obj))))
```

Model of TRAV. The TRAV model is:

X	execution of traversal through object graph (visit the objects specified by traversal spec)
X_{JP}	arrival at each object along the traversal
A	class and field declarations
A_{ID}	class names and complete field signatures[9]
A_{EFF}	provide reachability information
B	traversal specification and visitor
B_{ID}	traversal specification, *overloaded visitor methods*
B_{EFF}	call visitor and continue traversal (or not)

X is the actual traversal, and is implemented by `trav:weave`. Within that process, a call to `arrive` corresponds to a join point. This is analogous to the way, in the PA model, that a dynamic call to `call-method` corresponds to a join point.

p_A and p_B are clearly modeled in the implementation – p_A is the class and field declarations, which are converted into a class graph (bound to the `*classes*` global variable) by `eval-program` before executing `trav:weave`. p_B is the three arguments to `trav:weave`.

In A, the class names and field signatures are A_{ID}. The effect of A is to provide reachability information, which is modeled by the `reachable?` procedure. This combined with the traversal specification determines where the traversal goes. This combination happens in the `match?` procedure.

In B, B_{ID} is the traversal specifications, and is modeled by the `match-path` procedure. Combined with A_{EFF}, B_{EFF} determines whether to continue traversal, which is realized by the simple conditional branch on the result of the `match?` procedure.

4.3 COMPOSITOR – Class Composition

Hyper/J provides mechanisms that compose programs. This allows the programmer to implement concerns as independent (partial) programs, even when the composition of the concerns cuts across their module structure.

[9] Similar to a complete method signature, a complete field signature includes the class name, and is of the form: `<class> <enclosing-class>.<id>`

In this paper, we focus on the composition of classes. We omit class hierarchy composition, slicing based on concern maps and other powerful features of Hyper/J. For simplicity, we also limit ourselves to only a simple composition semantics that merges two programs based on class and member names. Using our simplified semantics, called COMPOSITOR, the display updating functionality from Section 4.1 can be implemented in two steps as follows. First we write a program with just this class:

```
class Observable {
  Display display;
  void moved() {
    display.update(this);
  }
}
```

calling the original figures program of Section 2.1 program A, and this one program B, the two programs are composed with a call to the compositor (weaver) as follows:

```
(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")
```
[10]

In the resultant composed program, the specified methods of the `Point` and `Line` classes are combined with the body of the `moved` method above. The effect is that they call `display.update` after they finish executing.

Implementation of COMPOSITOR. The weaver for COMPOSITOR is a source-to-source translator, which merges two BASE programs into one, under control of a composition description. The code for the weaver is

```
(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
```

[10] The actual Hyper/J meta-program for this composition would look something like:

```
mergeByName;
bracket "{Point,Line}"."set*"
  after Observable.moved($OperationName);
```

But for simplicity, we do not implement pattern matching and bracketing mechanisms. Instead, we assume that two methods with the same name match regardless of the parameter types, and that when methods from programs A and B are merged, the bodies of those methods are placed in A, B order in the merged method.

```
                              relationships)))
                  (loop (add-decl-to-pgm decl pgm
                                          signature)
                        (remove-subsets jp (cdr seeds))))
               (loop pgm (cdr seeds))))
         pgm))))
```

It receives two programs, `pgm-a` and `pgm-b`, as well as the description of the matching and merging to use, `relationships`.

The first step is to compute all possible compositions of declarations in the merged program. We call these seeds, and they are produced by `compute-seeds`. We model this as computing the power set of the union of the declarations in `pgm-a` and `pgm-b` (of course our implementation does not actually compute the power set). The list of seeds is sorted in set-inclusion order, with subsets following supersets.

After sorting, for each seed, there are up to three steps:

1. The procedure `all-match` determines whether this set should actually be merged according to the composition description, and returns the signature for the composed declaration when it should. In the simplest case, a seed of two method declarations (coming from `pgm-a` and `pgm-b`) having the same signature *m* matches, and returns *m* as the signature for the composed declaration. I.e. the set:

   {`<Point.setX(int)>`, `<Observable.moved()>`}

   is merged to `<Point.setX(int)>` in the composed program.
2. The procedure `merge-decls` computes the body of the actual declaration. I.e.
   ```
   {
     this.x = x;
     display.update(this);
   }
   ```
3. The procedure `add-decl-to-pgm` adds the declaration to the resultant program with the computed signature.

For any seed that is merged, all subsets of that seed are removed from the remaining seeds before proceeding.

The procedure `all-match` first picks a signature from the declarations in the given seed, and then checks that all the declarations can contribute to the signature. In order to allow renaming, it takes a relationships parameter as an additional argument:

```
(define all-match
  (lambda (decls relationships)
    (let ((sig (pick-signature decls
                               relationships)))
      (and (every? (lambda (decl)
                     (signature-match? sig decl
                       relationships))
                   decls)
           sig))))
```

The `merge-decls` procedure also receives relationships as an argument. Richer merging mechanisms, such as overriding and bracketing (i.e., before/after/around-like merger), can be supported by extending the meta-language and this procedure.

Model of COMPOSITOR. The description of the COMPOSITOR model is as follows. Note that unlike the other mechanisms, A and B are in the same language, and in this case we also use the META parameter.

X	the composed program
X_{JP}	declarations in X
A, B	class, method and field declarations
A_{ID}, B_{ID}	class, method and field signatures
A_{EFF}, B_{EFF}	provide declaration
META	rules for matching *and merging*

In the code, p_A and p_B are modeled as separate parameters to `compositor:weave`. p_{META} is the third parameter to this procedure. A and B are treated equally in the code, and can easily be generalized to a list of programs for composing more than two programs. X is the resultant composed program, which initially is empty, and is populated with declarations during the weaving process.

Join points are modeled as declarations in X. Each one corresponds to the merging of a subset of declarations from p_A and p_B. So seeds are in fact seeds for join points. If they match the match/merge description (p_{META}) they are merged to form an actual join point. Note that a single declaration from A or B can contribute to more than one declaration in X and vice versa.

A_{ID}, B_{ID} is the signatures of the declarations in A and B. The matching rules A_{ID}, B_{ID} work with are modeled by the META argument to the weaver.

A_{EFF}, B_{EFF} is simply to contribute the declaration from A or B to the merge. The actual merging is controlled by META.

4.4 OC – Open Classes

Open class mechanisms make it possible to locate method or field declarations for a class outside the textual body of the class declaration. Open classes are used in a variety of ways to modularize code; a common use is in visitor problems.

In this section, we work with a simple version of open classes in which method declarations are contained within class declarations, but it is possible to mark certain method declarations as defining methods on another class. We call this simple semantics OC.

Building on the running example, the following OC code defines draw methods for the different kinds of figure elements in a single `DisplayMethods` class – it modularizes the display aspect of the system.

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

Implementation of OC. We implement OC as a pre-processor that operates on a program consisting of normal BASE code intermixed with open class method declarations and produces a BASE program. This pre-processor pass is defined as

```
(define oc:weave
  (lambda (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-ocmdecls)))))
             (pgm-class-decls pgm))))))
```

The first step is to remove all open class method declarations from the input program. This is done by `remove-oc-mdecls` and `gather-oc-mdecls`. The open class method declarations are then each copied into their appropriate class. The `lookup-oc-mdecls` procedure finds, for a given class name, which open class method declarations should be copied into it. The `copy-oc-mdecls` procedure then copies those declarations, changing their signature from the open class form `<cname>.<mname>` to the normal BASE form `<mname>`.

```
(define lookup-oc-mdecls
  (lambda (cname all-oc-mdecls)
    (collect-if
      (lambda (oc-mdecl)
        (eq? (oc-mdecl-cname oc-mdecl) cname))
      all-oc-mdecls)))

(define copy-oc-mdecls
  (lambda (cname per-class-oc-mdecls)
    (map (lambda (oc-mdecl)
           (make-method-decl cname
             (oc-mdecl-rtype  oc-mdecl)
             (oc-mdecl-mname  oc-mdecl)
             (oc-mdecl-params oc-mdecl)
             (oc-mdecl-body   oc-mdecl)))
         per-class-oc-mdecls)))
```

Model of OC. The description of OC in terms of the model is as follows:

X	combined program
X_{JP}	class declarations
A	class declarations without OC method declarations

A_{ID}	effective method signatures (cname from enclosing class declaration)
A_{EFF}	method declaration stays in place
B	OC method declarations
B_{ID}	effective method signatures (cname from OC method declaration)
B_{EFF}	copy method declaration to target class

X is modeled as the results of the `oc:weave` procedure. p_A and p_B are a BASE program stripped of open class method declarations and the sets of open class method declarations respectively. The join points are the class declarations in the result program. A_{ID} happens by inclusion – the normal methods in p_A are copied into their same enclosing classes in X. This is the same effect as saying that the complete signature of methods in A is A_{ID}. B_{ID} is also the complete signature, which is explicit in B. The matching process for B is modeled by `lookup-oc-mdecls`. A_{EFF}, B_{EFF} are the same, the method declaration is copied into the class it belongs in X.

5 Restoring Functionality

To have confidence in the applicability of the modeling framework, we must be sure that in our simplified semantics – PA, TRAV etc. vs. AspectJ, Demeter etc. – we did not leave out issues that cannot be captured by the framework. This section addresses that concern by showing how several key missing functionalities could be added without falling outside the scope of the models.

There are two ways to show this, the strongest is to show that the actual models of each semantics change only in their details. The second is to show that even though a new model is required, it still fits within the same framework. In all the cases below, we show the former. We show this by once again appealing to the implementation and using it to intuitively validate that the model changes are only minor. Since all these changes are highly localized in the implementation, we claim they do not change the deep model structure.

5.1 Adding features to PA

The PA semantics is missing several key features of AspectJ, including additional kinds of join points, before and around advice, and context-sensitive pointcuts like `cflow`.

To add more kinds of join point, we must enrich the space of `jp` structures, and have more places in the interpreter perform advice lookup and execute operations. For example, join points for reading a field could be added to PA by defining a structure as follows:

```
(define-struct (get-jp jp) (fname))
```
[11]

and replacing the body of `get-field` just as Section 4.1 does for `call-method`.

[11] Assume that we first define a structure type `jp`, and modify `call-jp` to be a sub-type of it.

To add additional pointcuts (excluding cflow-like pointcuts), we simply extend pointcut-matches to identify join points matching those pointcuts. This could include pointcuts that identify only one kind of join point, such as call and get as well as pointcuts like target that identify multiple kinds of join points. In some cases implementing a new pointcut can require that additional information be added to some or all kinds of join point. For example, adding a within pointcut would require adding information about the lexically enclosing method to call join points.

To add before or around advice, we modify execute-advice to run the pieces of advice and call the thunk in the appropriate order. Supporting proceed can be done in a manner similar to super calls. We modify execute-advice to make a lambda closure for the remaining processing at the join point and put the closure in the environment of around advice execution. We also extend eval-exp to extract the closure from the environment and execute it for proceed.

Allowing proceed to change the arguments that inner advice and the method receive is more complicated. It requires changing the thunk passed to execute-advice to take a single argument, args, which is a list of those arguments. When a proceed is evaluated, the values of its operands are passed to the closure for proceed, which eventually passes them to the thunk.

To add control flow sensitive pointcuts like cflow, we thread a call stack through the join point structures. This is done by adding a stack-prev-jp field to all jp structures. This field holds the previous join point on the call stack. This requires the code that constructs the join points to keep the last top of stack and thread it properly. We can do this with the fluid-let mechanism in Scheme:

```
(define call-method
  (lambda (mname obj args)
    (let* ((jp (make-call-jp .stack-previous-jp.
                             mname obj args))
           ...)
      (fluid-let ((.stack-previous-jp. jp))
        ...))))
```

Here .stack-previous-jp. is effectively a dynamically scoped variable. The new cflow clause of pointcut-match follow the stack-prev-jp field until it either finds a matching jp or reaches the bottom of the stack.

A frequently-proposed feature for PA like mechanisms is to add an attribute feature to method declarations [1], and allow pointcuts to identify join-points based on these attributes [26]. This feature can easily be added to PA. Doing it for method declarations and call join points requires extending the language syntax to support attributes, extending join point structures to include an attribute element, modifying call-method to include the attribute in the join point, and adding a new kind of pointcut to match based on attributes.

These changes add detail to the previous model for PA, but they do not change its structure. Field gets are, like method calls, points in the flow of execution. Similarly the new kinds of pointcuts are no more than that – new kinds of pointcuts. Before and around advice require changes only to execute-advice. Adding proceed with arguments is only a little less localized.

5.2 TRAV

Demeter, DemeterJ and DJ differ slightly in terms of whether the traversal specification itself has control over whether the visitor is called. In DJ, for example, this is controlled by whether the visitor has an overloaded method for the type of visited object. This range of behaviors can be modeled in the implementation of `call-visitor`. To do so we adopt a naming convention that simulates overloading; then, before calling a visitor on an object, the traverser checks whether the visitor has a method for the class of the object, and then calls that method if it exists:

```
(define call-visitor
  (lambda (visitor obj)
    (let ((mname (visitor-mname (object-cname obj))))
      (if (has-method? (object-cname visitor) mname)
          (call-method mname visitor (list obj))))))
```

Again, these changes are local in the model implementation, and affect only details of the TRAV model.

5.3 COMPOSITOR

Hyper/J provides a rich meta-language that controls the composition, namely, bracketing (inserting method bodies before, after, or around of another method body), overriding, renaming, and wild-carding for matching. These can be supported by extending `all-match` and `merge-decls`. Since the enriched meta-languages can specify different merging strategies for different declarations contributing to a join point, `all-match` has to return both a matched signature and a list of merging directives for each matched declaration, so that `merge-decls` can know what to do.

Once again, these changes do not affect the basic structure of the COMPOSITOR model, as evidenced by the way they are localized in the implementation.

6 Modular Crosscutting

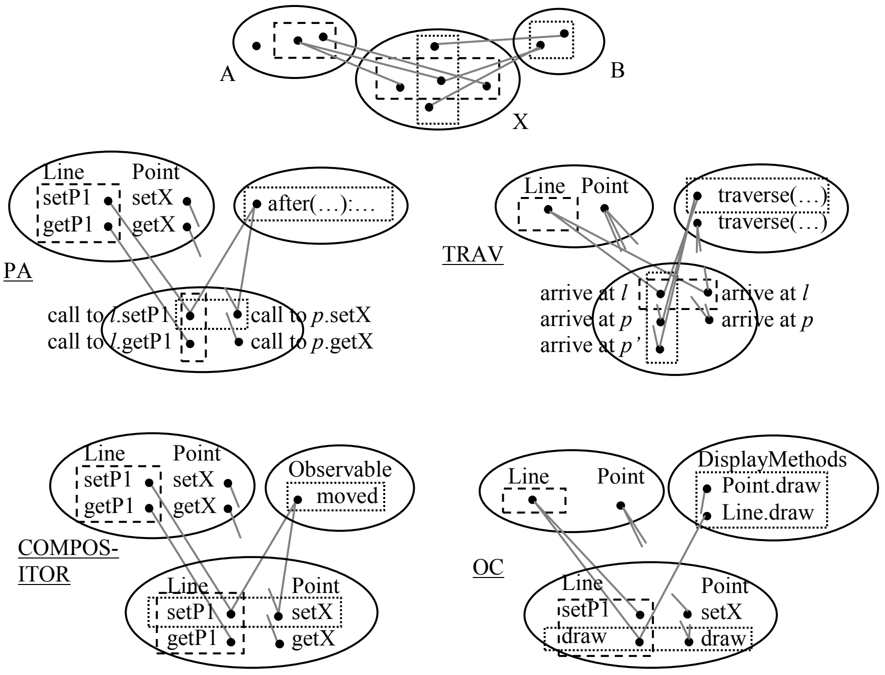
Our models provide a basis for understanding and comparing how each of the mechanisms enables crosscutting modularity. To do so we first need to address the A_{MOD} and B_{MOD} model parameters. These are the units of modularity in the A and B languages. We define these as follows:

	A_{MOD}	B_{MOD}
PA	class	advice
TRAV	class	traversal specification
COMPOSITOR	class	class
OC	class	class

These are not the only possibilities for units of modularity in each of these models. In PA for example, we could do as AspectJ does, and B_{MOD} could be aspect declarations.

In all the models we could use higher-level units of modularity like packages. The analysis of this section can be repeated for such alternative units of modularity to compare how crosscutting is supported for each.

For a module m_A (from p_A) we say that the *projection* of m_A onto X is the set of join points identified by the A_{ID} elements within m_A. The same is true for m_B. For example, in PA, the projection of a given advice declaration is all the join points matched by the pointcut of that advice declaration.



The dots in X represent JPs, the dots in A/B represent elements that match those JPs, the dashed/dotted boxes in A/B represent modules, and the dashed/dotted boxes in X are the projections of those modules. In all these cases the modules in A and B crosscut with respect to X.

Fig. 1. Modular Crosscutting in General Terms and in the Four Example Programs

For a pair of modules m_A and m_B (from p_A and p_B) we say that *m_A crosscuts m_B with respect to X* if and only if their projections onto X intersect, and neither of the projections is a subset of the other. Fig. 1 illustrates this situation, first in general terms and then for each of the example programs as explained below.

In the PA example (Section 4.1) the `Point` class and the display updating advice crosscut each other with respect to X. We consider the projection of the `Point` class onto X to include all calls to methods of the `Point` class.[12] The projection of the

[12] In AspectJ, the call join points would be considered not as being within the projection of the class, but rather as being within the projection of the calling class. Execution join points would be considered as within the projection of the class that defines the method.

advice onto X includes calls to the setter methods of the `Point` and `Line` classes. So the projections of the `Point` class and the advice intersect, and neither is subset of the other.

In the TRAV example (Section 4.2) the `Point` class and the traversal description crosscut with respect to X. The projection of the `Point` class includes all arrivals at `Point` objects initiated by any traversals. The projection of the traversal description includes arrivals at `Point` objects as well as arrivals at objects of other classes such as `Line` and `LinkedList`.

In the COMPOSITOR example (Section 4.3) the `Point` class and the `Observable` class crosscut with respect to X. The projection of the `Point` class on X includes all methods of the `Point` class in X. The projection of the `Observable` in B includes all set methods of `Point` and `Line` classes in X.

In the OC example (Section 4.4) the `Point` class in A and the `DisplayMethods` class in B crosscut with respect to X. The projection of `Point` in A contains all but the `draw` method of `Point` in X, and the projection of `DisplayMethods` contains draw methods of `Point` and `Line` classes.

Note that this analysis does not allow us to say that a given mechanism is crosscutting, only that it can support modular crosscutting. Or, in model terms, we cannot say a model is crosscutting, just that a particular pair of modules from particular p_A and p_B crosscut each other. This is not surprising, we know that an OO program does not have to have a hierarchical inheritance structure; it is simply that the language supports it.

Stepping back from the examples and the details of the models, we can see a clear three part characterization of what is required to support crosscutting structure: a common frame of reference that two (or more) programs can use to connect with each other and each provide their semantic contribution. In model terms the common frame of reference is X_{JP}, the programs are p_A and p_B, they connect at the join points using A_{ID} and B_{ID}, and provide their semantic contribution with A_{EFF} and B_{EFF}.

In some mechanisms, including PA, it can be tempting to equate the frame of reference with one of the programs, i.e. to say that the classes are X and that member declarations in the classes are the join points. But this two part characterization is less general It is difficult to model COMPOSITOR semantics or to model more than two crosscutting modules that way. The three part model supports both of these cleanly.

7 Related Work

Some authors have compared two particular AO systems, for example, Lieberherr et al. have explained concepts in the Demeter systems in terms of AspectJ [14]. Our work defines a common framework in terms of which four systems are modeled.

Some authors have proposed formal models for AO mechanisms [2, 6, 13, 20, 27]. Those models are attractive in that they express deep characteristics of the systems, such as implementation issues [17] and static analysis [25], in concise ways. But they only apply to specific mechanisms, all of which are in the PA family. Again, our work differs in finding common structure for diverse AO mechanisms. We believe that our framework could also be useful in developing other kinds of models for specific mechanisms. In fact, Wand's semantics[27] is based on our earlier work on the modeling framework. Note that this is in contrast to characterizations of weaving that work only in terms of source code pre-processing.

Filman and Friedman suggest that AOP systems can be defined as enabling quantified programmatic assertions over programs written by programmers oblivious to such assertions [7]. The model they propose is a two part-model – it compares programs and assertions. Our three-part structure is essential to accounting for how the mechanisms enable modular crosscutting, a key goal identified in [12]. Our framework can also model mechanisms that involve less "obliviousness", in that it can describe mechanisms such as attribute-based pointcuts described in Section 5.1.

8 Future Work

Further development of these models is one area of future work. For example, it appears that a variant of the PA model should apply to systems like Composition Filters [3, 4], and aspect-oriented frameworks[24].

We would like to enhance the model, and the implementations, to have a single parameterized weaving process. In the models, the weaving process consists of three operations: generate a join point, use A_{ID} and/or B_{ID} to identify elements in p_A and/or p_B matching the join point, and use A_{EFF} and/or B_{EFF} to produce the proper effects from the matching elements. The following code shows these steps:

```
(lambda (pA pB)
  ...
  (let ((jp <generate a JP>))
     ...
     (apply-A (lookup-A jp ...pA...))
     ...
     (apply-B (lookup-B jp ...pB...))
     ...))
```

But differences among the models make it difficult to actually implement all four using a parameterizable procedure of this form. These differences include:

- COMPOSITOR generates candidate join points rather than actual join points.
- B_{EFF} in PA takes as input a thunk that does lookup and apply for A. This makes it possible for B_{EFF} to control execution of A.
- The lookup and apply for A is implicit in OC. COMPOSITOR has a folded lookup and apply for A and B. In TRAV, A_{ID} and B_{ID} work together on each join point.

While a single parameterized weaving process is attractive, not having it does not seem to be a cause for significant concerns. Similar modeling frameworks for OO tend to work in terms of common terms more than a single parameterized implementation. The interpreters from [8] on which our code is based are not, for example, parameterized or composable.

9 Summary

Developing a common set of concepts with which to discuss and compare AO mechanisms is a critical task. This paper takes a step in this direction by presenting a set of models that can be used to compare how different AOP mechanisms provide support for modular implementation of crosscutting concerns. The analysis yields a three-part characterization of what is required for two programs coordinate their semantic contributions in terms of a common frame of reference. Modules in the programs are said to crosscut each other with respect to the frame of reference. This simple model makes it possible to capture all four mechanisms, and expands naturally to cover more than two crosscutting modules.

Acknowledgements

The authors are grateful to Kris de Volder, Mitchell Wand, Chris Dutchyn, Gail Murphy, Harold Ossher, Karl Lieberherr, Mira Mezini, and Mario Südholt for their valuable comments and discussion on early drafts of the paper. Previous work with Wand and Dutchyn developed earlier versions of the Aspect Sandbox.

References

1. C# Language Specification (2nd edition), ECMA Standard-334, 2002.
2. Andrews, J., Process-Algebraic Foundations of Aspect-Oriented Programming. In *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, LNCS 2192, Springer, pp.187-209, 2001.
3. Bergmans, L. and Aksit, M. Composing Crosscutting Concerns Using Composition Filters. *Communications of ACM*, *44*(10)51-57, 2001.
4. Bergmans, L., Aksit, M. and Tekinerdogan, B. Aspect Composition Using Composition Filters. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer, pp.357-382, 2001.
5. Clifton, C., Leavens, G., Chambers, C. and Millstein, T., MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Conference on Object Oriented Programming Systems Languages and Applications*, pp.130-145, 2000.
6. Douence, R., Motelet, O. and Südholt, M. A Formal Definition of Crosscuts. In *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, LNCS 2192*, Springer, pp.170-186, 2001.
7. Filman, R. and Friedman, D. Aspect-Oriented Programming is Quantification and Obliviousness. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns*.
8. Friedman, D., Wand, M. and Haynes, C.T. *Essentials of Programming Languages*. MIT Press, 2001.
9. Kelsey, R., Clinger, W. and Rees, J. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, *11*(1)7-105, 1998.

10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. Getting Started with AspectJ. *Communications of ACM*, *44*(10)59-65, 2001.
11. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. An overview of AspectJ. In *European Conference on Object-Oriented Programming, LNCS 2072*, Springer, pp.327-353, 2001.
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. Aspect-Oriented Programming. In *European Conference Object-Oriented Programming, LNCS 1241*, Springer, pp.220-242, 1997.
13. Laemmel, R. A Semantical Approach to Method-call Interception. In *International Conference on Aspect-Oriented Software Development*, pp.41-55, 2002.
14. Lieberherr, K., Orleans, D. and Ovlinger, J. Aspect-Oriented Programming with Adaptive Methods. *Communications of ACM*, *44*(10)39-41, 2001.
15. Lieberherr, K. *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns*. PWS Publishing, 1996.
16. Lieberherr, K. and Orleans, D. Preventive Program Maintenance in Demeter/Java (Research demonstration). In *International Conference on Software Engineering*, pp.604-605, 1997.
17. Masuhara, H., Kiczales, G. and Dutchyn, C., A Compilation and Optimization Model for Aspect-Oriented Programs. In *International Conference on Compiler Construction*, LNCS 2622, Springer, pp.46-60, 2003.
18. Millstein, T. and Chambers, C., Modular Statically Typed Multimethods. In *European Conference on Object-Oriented Programming*, LNCS 1628, Springer, pp.279-303, 1999.
19. Moon, D., Object-oriented programming with Flavors. In *Conference on Object Oriented Programming Systems Languages and Applications*, pp.1-8, 1986.
20. Orleans, D. Incremental Programming with Extensible Decisions. In *International Conference on Aspect-Oriented Software Development*, pp.56-64, 2002.
21. Orleans, D. and Lieberherr, K. DJ: Dynamic Adaptive Programming in Java. In *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns,* LNCS 2192, Springer, pp.73-80, 2001.
22. Ossher, H. and Tarr, P. Hyper/J: Multi-dimensional separation of concerns for Java. In *International Conference on Software Engineering*, pp.729-730, 2001.
23. Ossher, H. and Tarr, P. The Shape of Things To Come: Using Multi-Dimensional Separation of Concerns with Hyper/J to (Re)Shape Evolving Software. *Communications of ACM*, *44*(10)43-50, 2001.
24. Pinto, M., Fuentes, L., Fayad, M. and Troya, J. Separation of Coordination in a Dynamic Aspect Oriented Framework. In *International Conference on Aspect-Oriented Software Development*, pp.134-140, 2002.
25. Sereni, D. and de Moor, O., Static Analysis of Aspects. In *International Conference on Aspect-Oriented Software Development*, pp.30-39, 2003.
26. Shukla, D., Fell, S. and Sells, C. Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. *MSDN Magazine*, *March*, 2002.
27. Wand, M., Kiczales, G. and Dutchyn, C. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. In *Foundations of Aspect-Oriented Languages (FOAL2002)*, pp.1-8, 2002.
28. Weinreb, D. and Moon, D.A. Flavors: Message passing in the LISP machine, A.I. Memo 602, Massachusetts Institute of Technology A.I. Lab., 1980.