A Parameterized Interpreter for Modeling Different AOP Mechanisms

Naoyasu Ubayashi
Department of Artificial Intelligence
Kyushu Institute of Technology
Fukuoka, Japan
ubayashi@acm.org

Genki Moriyama
Department of Artificial Intelligence
Kyushu Institute of Technology
Fukuoka, Japan
genki@acm.org

Hidehiko Masuhara
Graduate School of Arts and Sciences
University of Tokyo
Tokyo, Japan
masuhara@acm.org

Tetsuo Tamai
Graduate School of Arts and Sciences
University of Tokyo
Tokyo, Japan
tamai@acm.org

ABSTRACT

We present a parameterized interpreter for modeling aspect-oriented mechanisms. The interpreter takes several parameters to cover different AOP mechanisms found in AspectJ, Hyper/J, and Demeter. The interpreter helps our understanding of the AOP mechanisms in two ways. First, its core part represents the common mechanical structure shared by different AOP mechanisms. Second, by reconstructing the existing AOP mechanisms and using parameters to configure the interpreter, we can illustrate the differences and similarities of those mechanisms clearly. This will also be helpful in rapid-prototyping a new AOP mechanism or a reflective AOP system that supports different mechanisms.

Categories and Subject Descriptors: D.3.2 Programming Languages: Language Classifications –*Extensible languages*

General Terms: Languages

Keywords: AOP, Join point models

1. INTRODUCTION

Mechanisms in aspect-oriented programming (AOP) languages[14][8] can be characterized by join point models (JPMs) consisting of join points, a means of identifying the join points, and a means of raising effects at the join points. JPMs are important in AOP languages because they can deal with crosscutting concerns elegantly[30]. Crosscutting concerns may not be able to be modularized as aspects without an appropriate JPM. Each of the current AOP languages is based on a few fixed set of JPMs. Many different JPMs have been proposed, and they are still evolving with the aim of better modularization of various crosscutting concerns.

H.Masuhara and G.Kiczales defined a three-part modeling framework that explains a common structure in different JPMs including PA (pointcuts and advice as in AspectJ[2][15]), TRAV (traversal specifications as in Demeter[6]), COMPOSITOR (class merges based on matching relationships as in Hyper/J[27][22]), and OC (open classes as in AspectJ)[19]. The modeling framework is derived from a suite of interpreters called Aspect SandBox (ASB)[1][7].

Although the three-part modeling framework clarifies common mechanisms in major JPMs, it does not provide a common design model. The goal of this paper, a follow-up paper to [19], is a conceptual description of the design space of JPMs, executed by capturing essential characteristics and differences concisely. Based on the three-part modeling framework, we propose a parameterized interpreter that takes several parameters to cover different JPMs.

The interpreter helps our understanding of the AOP mechanisms in two ways. First, its core part represents the common mechanical structure shared by different JPMs. Second, by reconstructing the existing JPMs and using parameters to configure the interpreter, we can illustrate the differences and similarities of those mechanisms clearly. This will also be helpful in rapid-prototyping a new JPM or a reflective AOP system that supports different mechanisms.

The remainder of this paper is structured as follows. In section 2, we point out the difficulty of providing a common parameterized interpreter by using examples that show the differences among the four JPMs. We overcome the difficulty, and present the parameterized interpreter in section 3. We also show how the four JPMs are obtained by fulfilling the parameters of the interpreter. In section 4, we evaluate the parameterized interpreter in terms of the efficiency of rapid-prototyping JPMs. In section 5, we discuss extensible AOP using the parameterized interpreter. In section 6, we introduce some related works, and discuss future directions of research. Section 7 concludes the paper.

2. MOTIVATION

Although the previous work pointed out the common structure shared by different AOP mechanisms[19], the commonality is described in an informal manner. In other words, there has been no single model to capture different AOP mechanisms. Note that, as in the previous work, we refer to an interpreter of a language with an AOP mechanism as a model of the mechanism. In this section, we briefly excerpt a modeling framework with sample programs from the previous work, and then demonstrate how the framework lends itself to no single model.

2.1 Example

Using the following simple figure program, we give a brief explanation of the four JPMs:

```
class Figure { List element = new LinkedList(); }
class FigureElement { Display display; }
class Point extends FigureElement {
  int x, 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, p2;
  int getP1() { return p1; }
  int getP2() { return p2; }
  void setP1(Point p1) { this.p1 = p1; }
  void setP2(Point p2) { this.p2 = p2; }
}
```

This program is written in the ASB core language called BASE[1]. In ASB, interpreters of the four JPMs are designed on the BASE interpreter. The above program consists of four classes: `Figure`, `FigureElement`, `Point`, and `Line`. A figure is comprised of a collection of figure elements. There are two kinds of figure elements: point and line. The definitions of the two classes `LinkedList` and `Display` are omitted here. The former is a class for managing figure elements, and the latter is a class for displaying a figure.

PA program

PA captures a join point such as a method call, and inserts advice code before/after/around the join point. The following code is an after-advice that implements display updating functionality. The `update`, which is the method of the `Display` class, is called after `setX`, `setY`, `setP1`, or `setP2` is called.

```
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);
}
```

TRAV program

TRAV provides a mechanism that enables programmers to design traversals through object graphs in a succinct fashion. The following code implements the behavior of visiting all the figure elements reachable from a figure.

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

[1]While this is a Scheme-based object-oriented language, we use a Java-like syntax for readers' easier understanding.

The first argument to `traverse` is called a traversal specification that describes a path to be visited. The second argument is the root object where the traversal starts. The third argument is a visitor that defines behavior at each traversed object. In this case, the program traverses from a root figure object following down through line objects to reach point objects, and counts the elements.

COMPOSITOR program

COMPOSITOR composes independent partial programs. Using COMPOSITOR, the display updating functionality can be designed in two steps. First, we write the following program and relationship:

```
class Observable {
  Display display;
  void moved() { display.update(this); }
}
; relationship between Point/Line and Observable
match Point.setX with Observable.moved
match Point.setY with Observable.moved
match Line.setP1 with Observable.moved
match Line.setP2 with Observable.moved
```

Next, we compose the original figure program and the above program using the relationship. In the resulting composed program, the specified method of the `Point` and `Line` classes are combined with the body of the `moved` method in the `Observable` class. The effect is that `display.update` is called after the execution is complete.

OC program

OC makes it possible to locate method or field declarations for a class outside the textual body of the class declaration. The following code defines `draw` methods for the different kinds of figure elements in a single `DisplayMethods` class – it modularizes the display aspect of the system. `Graphics` is a class for drawing graphics.

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

2.2 Three-part modeling framework

Although the four JPMs are drastically different, there are some points of commonality. The three-part modeling framework shows the core semantics of these JPMs by modeling the weaving process. The framework defines the process of weaving as taking two programs and coordinating them into a single combined computation. A critical property of the framework is that it describes the join points as existing in the result of the weaving process rather than residing in either of the input programs.

The framework explains each JPM as an interpreter that is modeled as a tuple of nine parameters:

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

A and B are the languages in which the respective programs p_A and p_B, input to the interpreter, are written. X is the result domain of the weaving process, which is the third language of a computation. X_{JP} is a join point in X. A_{ID} and B_{ID} are the means, in the languages A and B, of identifying elements of X_{JP}. A_{EFF} and B_{EFF} are the means, in the languages A and B, of affecting semantics at the identified join points. $META$ is an optional meta-language for

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

Table 1: Three-part modeling framework

parameterizing the weaving process. A weaving process is defined as a procedure that accepts p_A, p_B, and $META$, and produces either a computation or a new program. Table 1 summarizes the three-part modeling framework[19]. In Table 1, single letters 'c', 'm', and 'f' are abbreviations for class, method and field, respectively.

2.3 Problem to be tackled

In the three-part modeling framework, the weaving process consists of three operations: 1) generating a join point; 2) applying A_{ID} and B_{ID} to identify elements in p_A and p_B matching the join point; 3) using A_{EFF} and B_{EFF} to produce the proper effects from the matching elements. These steps are illustrated by the following code skeleton written in Scheme.

```
(lambda (pA pB)
  (let ((jp <generate a join point>))
    (effect-A (lookup-A jp pA))
    (effect-B (lookup-B jp pB))))
```

As pointed out in [19], the differences among the four JPMs make it difficult to design a single parameterized procedure of this form. For example, B_{EFF} controls execution of A in PA. As a consequence, the four JPMs are designed as individual interpreters in ASB.

Although the three-part modeling framework identified a common structure among different AOP mechanisms as shown in Table 1, the commonality is given in an informal manner. There has been no single model that captures all the different AOP mechanisms.

3. PARAMETERIZED INTERPRETER

To tackle the problem pointed out in section 2, we present a single model that captures different AOP mechanisms, in the form of a parameterized interpreter written in Scheme. The interpreter, extensible ASB (X-ASB), consists of the core part and various sets of parameters. The former represents the common mechanical structure of the four JPMs PA, TRAV, COMPOSITOR, and OC. The latter clarifies the differences and similarities of those JPMs. Each JPM can be obtained by providing parameters to the interpreter.

3.1 Core part and the sets of parameters

Table 2 and Figure 1 show the relation between the parameters of the three-part modeling framework and those of X-ASB. The elements enclosed by brackets are procedures. The X-ASB parameters provided as procedure signatures expose the programming interfaces for JPM developers.

Three-part model	X-ASB parameter
X	[eval-program], [computation-at-jp]
X_{JP}	[register-jp]
A	pgm-a
A_{ID}	[lookup-a]
A_{EFF}	[effect-a]
B	pgm-b
B_{ID}	[lookup-b]
B_{EFF}	[effect-b]

Table 2: X-ASB parameters

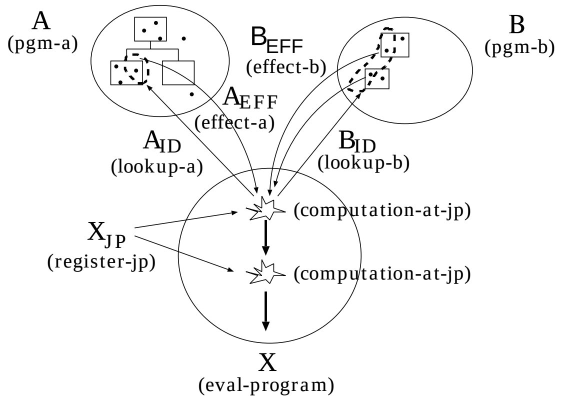

Figure 1: Weaving process

We illustrate the outline of the X-ASB parameterized interpreter. The heart of the weaving process is the coordination at join points where the two programs `A` and `B` meet. The type of join points is registered by the `register-jp` parameter. Each coordination given as the `computation-at-jp` parameter is executed using the four parameters `lookup-a`, `effect-a`, `lookup-b`, and `effect-b`. The `eval-program` parameter is the body of an interpreter in which procedures specified by the `computation-at-jp` parameter are called. The following code shows the core part of the interpreter.

```
(define weave
  (lambda (pgm-a pgm-b)
    (register-jp)
    (eval-program pgm-a pgm-b)))

(define eval-program
  (lambda (pgm-a pgm-b)
    ( <iterate the following steps
        - get the next program element
        - generate a join point
        - call computation-at-jp>)))
```

```
(define computation-at-jp
  (lambda (jp)
    <mediate the following according to the JPM type>
    (effect-a (lookup-a jp))
    (effect-b (lookup-b jp))))
```

The interpreter takes the two programs pgm-a/pgm-b as arguments that correspond to A/B in Table 1. The register--jp procedure registers join point types that include information needed for coordination at specific join points. For example, a method-call join point type (abbreviated as call-jp in this paper) is registered in PA. When the execution of the eval-program procedure arrives at a point that requires a weaving, it generates a join point instance from its type and calls the computation-at-jp procedure. In PA, the eval-program procedure evaluates the original program (pgm-a), generates a call-jp instance when the eval-program procedure evaluates the method-call expression, and calls the computation-at-jp procedure that executes an advice body (pgm-b) using information contained in the call-jp instance. The computation-at-jp parameter that handles the call-jp can be designed as follows.

```
;; The first version
(define eval-exp    ;called from eval-program
  (lambda (exp env)
    (cond
      ((method-call-exp? exp)
        (computation-at-jp::call-jp
          <generate an instance of call-jp>)))))
(define computation-at-jp::call-jp
  (lambda (jp)
    (execute-advice (lookup-advice jp
      (lambda ()
        (execute-method (lookup-method jp) jp))))))
```

The four procedures lookup-method, execute-method, lookup-advice, and execute-advice correspond to the X-ASB parameters lookup-a, effect-a, lookup-b and effect-b, respectively: lookup-method and lookup-advice searches the method declaration and the advice declarations related to the call-jp; execute-method executes the method body; and execute-advice executes the advice bodies. In this case, only the after-advice is available.

3.2 Registration of join point type

A join point type registered by the register-jp parameter is strongly related to the computation-at-jp parameter. For example, the computation-at-jp::call-jp procedure is affected by the call-jp join point. If we want to add a new kind of join point such as a field-set join point, we must define a new kind of computation-at-jp such as the computation-at-jp::fset. Although the first version of the computation-at-jp procedure shown above gives a guideline applicable to JPM designs, its reusability is still limited. In order to make the computation-at-jp procedure reusable, it is necessary to enrich the data structure that a join point holds. The following jp defines the structure of a join point:

```
(define-struct jp
    (computation-strategy
     lookup-a effect-a lookup-b effect-b))
```

The jp structure consists of five elements: the first element computation-strategy shows a weaving policy such as b-control-a (pgm-b controls over pgm-a as in PA); the four procedures lookup-a, effect-a, lookup-b, and effect-b correspond to parameters in Table 2. The elements specific to a certain join point type can be added to the jp structure. For instance, the call-jp join point in PA is defined below. This join point includes 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:

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

A set of join point types must be registered using register--jp. The following is the code for registering the call-jp join point type.

```
(define-struct jtype (jname generator))
(define register-jp
  (lambda ()
    (register-one-jp
      'call-jp
       (lambda (mname target args)
         (make-call-jp
           'b-control-a
            lookup-method execute-method
            lookup-advice execute-advice
            mname target args)))))
```

A join point type is defined by the jtype structure, which in turn has two elements: the jname element shows the name of the join point type, and the generator is a procedure that instantiates a join point from the jp structure. The register-one-jp procedure, an X-ASB library procedure, registers one join point type. Using the library procedure, it is possible to add a new kind of join point type and its related computation.

3.3 Computation at a join point

Using the jp structure, we give a new design of the computation-at-jp parameter. This is more modular than the first version.

When an interpreter arrives at a point specified by the registered join point type, the interpreter generates a join point instance using the generator defined in the jtype structure, and executes the following computation-at-jp procedure, which dispatches a process according to a computation strategy. In the case of the call-jp join point, the computation-at-jp::b-control-a library procedure is executed. It is not necessary to define a new kind of computation-at-jp whenever we add a new kind of join point type. We can reuse the computation-at-jp::b-control-a library procedure if the join point type is based on PA.

```
;; The Second version
(define computation-at-jp
  (lambda (jp)
    (let ((strategy (jp-computation-strategy jp)))
      (cond ((b-control-a? strategy)
              (computation-at-jp::b-control-a jp))
             ((traversal? strategy)
              (computation-at-jp::traversal jp))
             <other strategies are ommited>))))
(define computation-at-jp::b-control-a
  (lambda (jp)
    (let* ((lookup-a (jp-lookup-a jp))
           (effect-a (jp-effect-a jp))
           (lookup-b (jp-lookup-b jp))
           (effect-b (jp-effect-b jp)))
      (effect-b (lookup-b jp) jp
        (lambda ()
          (effect-a (lookup-a jp) jp))))))
```

	PA	TRAV	COMPOSITOR	OC
X	[eval-program]	[eval-program]	[eval-program]	[eval-program]
	[computation-at-jp]	[computation-at-jp]	[computation-at-jp]	[computation-at-jp]
X_{JP}	[register-jp]	[register-jp]	[register-jp]	[register-jp]
	call-jp	arrival-jp	matching-jp	c-decl-jp
A	c, m, f declarations	c, f declarations	c, m, f declarations	c declarations w/o OC declarations
A_{ID}	[lookup-method]	[lookup-fields]	[lookup-decl]	[lookup-cdecl]
A_{EFF}	[execute-method]	[provide-next-arrival]	[provide-decl]	[provide-mdecls]
B	advice declarations	traversal spec.&visitor	$(=A)$	OC m declarations
B_{ID}	[lookup-advice]	[lookup-trav]	$(=A_{ID})$	[lookup-oc-mdecls]
B_{EFF}	[execute-advice]	[execute-visitor]	$(=A_{EFF})$	[copy-mdecls]
$META$	B control A	traversal	relationship	oc

Table 3: Application of the X-ASB parameterized interpreter

The updated version of the `computation-at-jp` procedure is more reusable than the first version. This version can be commonly used by PA, TRAV, COMPOSITOR, and OC. The idea of join point type is essential to designing a JPM that is as modular as possible because information needed at the `computation-at-jp` parameter is encapsulated in a join point instance.

3.4 Library for pointcut designator

The pointcut mechanism is important and useful for effective AOP although all JPMs do not presume the mechanism. X-ASB provides library procedures for designing pointcut designators and pointcut evaluations.

The structure of a pointcut designator is defined as follows:

```
(define-struct pcd (pname evaluator))
```

A pointcut designator consists of two elements: the `pname` shows the name of a pointcut designator, and the `evaluator` is a boolean procedure that checks whether a current join point is an element of a pointcut set. The `evaluator` procedure is called from the `lookup-b` parameter.

Below is the code for registering a `call` pointcut designator that includes a type of return value, a class name, a method name, and parameters of the method. The `register-one-pcd` procedure, an X-ASB library procedure, registers one pointcut designator. The procedure has two arguments: the first is the name of the pointcut designator, and the second is an evaluator. In this case, the registered evaluator checks whether the method name of a join point is equivalent to the name specified by the `call` pointcut designator.

```
(define-struct (call-pcd pcd)
               (rtype cname mname params))
(define register-jp
  (lambda ()
    (register-one-pcd
      'call-pcd
      (lambda (ptc jp)
        (if (call-pcd? ptc)
            (and (eq? (call-pcd-mname ptc)
                      (call-jp-mname  jp))))))))
```

In PA, the `lookup-advice` procedure, which checks `call` pointcut conditions, uses the pointcut evaluator registered by the above code.

3.5 Design of the four JPMs

Table 3 shows the application of the X-ASB parameterized interpreter. Table 3 corresponds to the three-part modeling framework as follows: a parameter in the framework corresponds to a procedure that designs an X-ASB parameter, and the META parameter in the framework corresponds to the computation strategy in X-ASB. Although there are no META parameters in the framework except COMPOSITOR, the existence of these parameters is assumed implicitly in the framework. For example, the META parameter in PA can be regarded as the `b-control-a` computation strategy. The concept of a computation strategy is important to making the design of the `computation-at-jp` parameter reusable. Moreover, this concept resolves the problem pointed out in section 2.3.

The main contribution of this paper is to provide a process for designing JPMs. Using X-ASB, we can design JPMs explicitly as follows: 1) design a type of join point using the `register-jp` parameter, which includes the four parameters `lookup-a`, `effect-a`, `lookup-b`, and `effect-b`; 2) coordinate the computation at the join point using the `computation-at-jp` parameter; and 3) design a weaving process using the `eval-program` parameter in which the `computation-at--jp` is called. The registration of a join point type and coordination using the type are essential in the parameterized interpreter X-ASB. Based on X-ASB, interpreters of the four JPMs can be designed as follows (see Figure 2):

- PA: The `eval-program` procedure executes program pA, and calls the `computation-at-jp` procedure upon evaluating a method-call expression, a point related to the `call-jp` join point. The `computation-at-jp` procedure weaves the method execution of the program pA and the advice execution of the program pB using `lookup-method`, `execute-method`, `lookup-advice`, and `execute-advice`. These procedures are registered in the `call-jp` join point;

- TRAV: The `eval-program` procedure traverses program pA provided as an object graph, and calls the `computation-at-jp` procedure upon reaching a point related to the `arrival-jp` join point, an arrival at the node that satisfies the traversal specification. The `computation-at-jp` procedure weaves the program pA (class, field declarations) and the program pB (traversal specifications and visitor) by executing the visitor method that refers to the fields of the program pA. The `computation-at-jp` procedure uses the four pro-

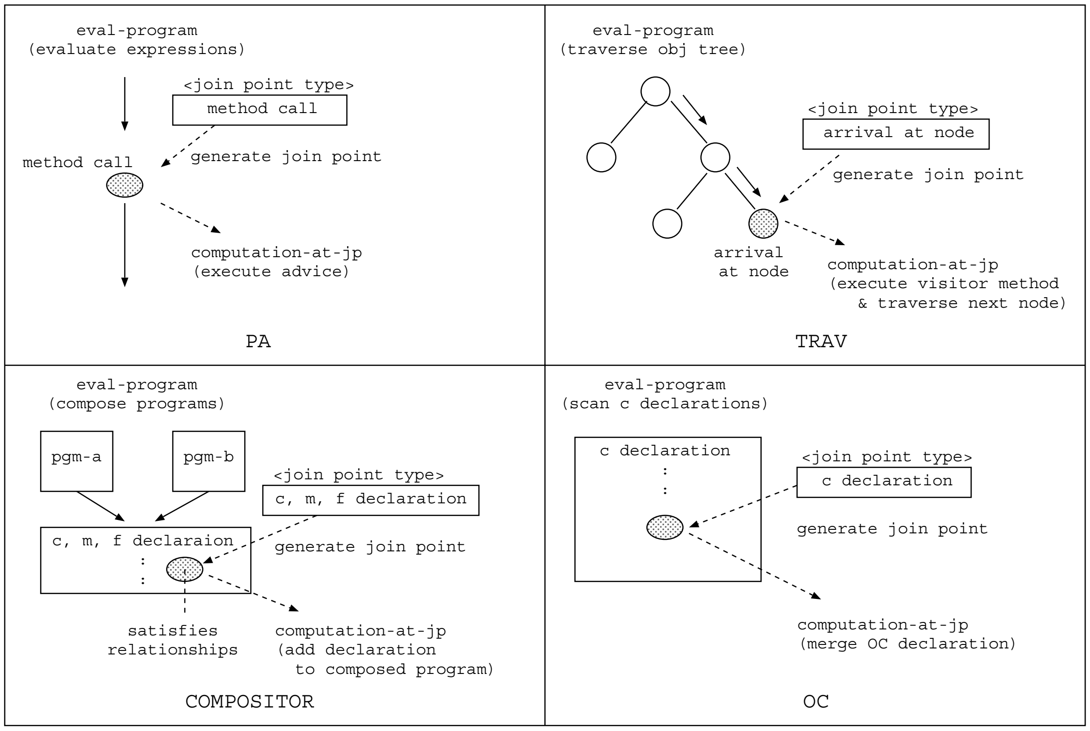

Figure 2: Design of the four JPMs

cedures `lookup-field`, `provide-next-arrival`, `lookup-trav`, and `execute-visitor`. These procedures are registered in the `arrival-jp` join point;

- COMPOSITOR: The `eval-program` procedure parses two programs *pA* and *pB* to find declarations that satisfy the relationship, and calls the `computation-at-jp` procedure upon reaching a point related to the `matching-jp` join point, a point matching the relationship. The `computation-at-jp` procedure weaves the program *pA* and the program *pB* by composing declarations using `lookup-decl` and `provide-decl`. The former looks up a declaration to be merged, and the latter adds the declaration to the merged program. These procedures are registered in the `matching-jp` join point;

- OC: The `eval-program` procedure parses program *pA* (class declarations), and calls the `computation-at-jp` procedure upon reaching a point related to the `c-decl-jp` join point, a class declaration matching OC declarations. The `computation-at-jp` procedure weaves the program *pA* and the program *pB* (OC declarations) by copying OC declarations to the related class declaration using `lookup-cdecl`, `provide-mdecl`, `lookup-oc-mdecls`, and `copy-mdecls`. These procedures are registered in the `c-decl-jp` join point.

For a detailed explanation, the program codes of the PA interpreter (example of dynamic JPM) and the COMPOSITOR interpreter (example of static JPM) are shown in the appendix as examples.

It is important for JPM developers to make the following design decisions: 1) What kinds of join points are needed? and 2) What kinds of coordination should be defined at the join points?

4. EVALUATION

X-ASB is effective for rapid-prototyping JPMs. In this section, we evaluate how rapidly we can model JPMs using X-ASB.

Table 4 shows the code size for developing the four JPMs. The code is categorized into four parts consisting of `register-jp` (and related parameters: lookup-a/b, effect-a/b), `computation-at-jp` (to be exact, a sub-procedure such as `computation-at-jp::b-control-a`), `eval-program`, and base. The base part includes the BASE interpreter and X-ASB library procedures except `computation-at-jp`. The code size `A`, the summation of the first three parts, indicates the LOC (Line of Code) specific to each JPM. The code size `B` shows the total LOC of each JPM weaver. The labor for introducing a new JPM can be estimated by the expression $A/B{*}100$. As shown in the table, we have only to add 10 - 30 % new code to develop a new JPM. Although the code for defining the `register-jp` parameter and the `computation-at-jp` parameter is relatively small excluding PA, the size of the `eval-program` is large. It is not necessarily easy to design this parameter because it determines the overall behavior of the weaver.

Next, we evaluate the labor for extending an existing JPM. Table 5 shows the code size for extending PA. We have only to add 48 LOCs and 42 LOCs in order to add fset-jp/fset-pcd (a field-set join point and pointcut designator) and fget-jp/fget-pcd (a field-get join point and pointcut designator), respectively. The labor for extending PA can be estimated by the expression $B/(A{+}B){*}100$. As shown in the table, we have only to add about 20 % new code to add

Part	PA	TRAV	COMPOSITOR	OC
1.register-jp	81	36	16	32
2.computation-at-jp::xx	9	16	5	11
3.eval-program	64	131	238	28
4.base	537	537	537	537
A: sum of 1-3	154	183	259	71
B: sum of 1-4	691	720	796	608
A/B (%)	22.3	25.4	32.5	11.7

Table 4: LOC for developing each JPM

Part	original PA	add fset-jp	add fget-jp
1.register-jp	81	44	38
2.computation-at-jp::xx	9	-	-
3.eval-program	64	4	4
sum of 1-3	A:154	B:48	B:42
B/(A+B) (%)		23.8	20.8

Table 5: LOC for extending PA

a new kind of join point and pointcut designator. The code of the `computation-at-jp` is reused. That is, the computation strategy `b-control-a` is common to `call-jp`, `fset-jp`, and `fget-jp`. It is easy to extend an existing JPM because we can reuse most of the two parameters `eval-program` and `computation-at-jp`.

5. DISCUSSION TOWARDS EXTENSIBLE AOP

In this paper, we proposed a parameterized interpreter for modeling different JPMs. Using this interpreter, we can introduce new kinds of JPMs. However, it would be better if we could change existing JPMs from base-level languages. The effectiveness in software evolution would be restricted if language developers had to extend JPMs whenever application programmers needed new kinds of JPMs. Extensible languages, such as computational reflection[24][18] and metaobject protocols[13] would be also useful in AOP.

Towards this direction, we discuss how metaobject protocols can be designed based on the parameterized interpreter.

5.1 JPM design layer

There are two layers for designing JPMs. The level 1 layer provides mechanisms for introducing new JPMs. The parameterized interpreter belongs to this layer. The level 2 layer provides mechanisms for extending JPMs developed in the level 1 layer. For example, in the level 2 layer, we can add a new pointcut designator to the PA interpreter developed in the level 1 layer. Reflection for AOP is realized in the level 2 layer.

Table 6 shows the relation between the design layers and parameters. The parameters `lookup-a/b` and `effect-a/b` are omitted because they are subjected to the `register-jp` parameter. The level 1 layer includes all the parameters that are necessary to design runnable interpreters. The level 2 layer includes only the `computation-at-jp` parameter and the `register-jp` parameter. Join point types, pointcut designators, and computation strategies (coordination at a join point) can be extended in the level 2 layer.

Layer	Purpose	Parameters
level 1	introduction of new JPMs	[eval-program] [computation-at-jp] [register-jp]
level 2	extension of existing JPMs	[computation-at-jp] [register-jp]

Table 6: JPM design layer

Metaobject protocol	Function
register-one-strategy	register a computation strategy
lookup-strategy	search a computation strategy
register-one-jp	register a join point type
lookup-jp	search a join point type
extract-jp	extract a join point information
register-one-pcd	register a pointcut designator
lookup-pcd	search a pointcut designator
extract-ptc	extract a pointcut information

Table 7: Metaobject protocols for AOP

From the evaluation in section 4, we can observe the following: it is relatively easy to design the `register-jp` parameter and the `computation-at-jp` parameter; and it is not easy to design the `eval-program` parameter. We believe that reflection for AOP should be limited to adding new kinds of join point types, pointcut designators, and computation strategies. That is, parameters such as `eval-program` should not be the target of reflection. Developing a new `eval-program` carries a cost equal to that of developing a new interpreter. Table 6 reflects this observation.

5.2 Metaobject protocols

Table 7 shows metaobject protocols for AOP. These protocols extend the two parameters `computation-at-jp` and `register-jp`. The three protocols `register-one-strategy` (register a computation strategy), `register-one-jp` (register a join point type), and `register-one-pcd` (register a pointcut designator) correspond to intercession. The five protocols `lookup-strategy` (search a computation strategy), `lookup-jp` (search a join point type), `extract-jp` (extract information of a join point instance), `lookup-pcd` (search a pointcut designator), and `extract-ptc` (extract a pointcut information) correspond to introspection. The `thisJoinPoint` variable in AspectJ corresponds to the `extract-jp` protocol. A reflective interpreter supporting the metaobject protocols is under construction.

Most aspect-oriented features can be designed by run-time metaobject protocols[25]. It is interesting to explore the relationships between reflective OOP languages and reflective AOP languages. We think that reflective AOP languages should expose metaobject protocols based on JPMs because they are essential to AOP.

5.3 Validity of the metaobject protocols

It is necessary to evaluate the validity of the metaobject protocols from the viewpoint of generality. We have examined how existing research on JPM extensions can be explained using these protocols. We believe that they are generic protocols that can explain almost all the current JPM extensions.

Table 8 shows the relation between existing studies and metaobject protocols. There are investigations that have

Metaobject protocol	Example
register-one-strategy	*none*
lookup-strategy	*none*
register-one-jp	*none*
lookup-jp	*none*
extract-jp	thisJoinPoint in AspectJ
register-one-pcd	pcflow, dflow, josh pattern-based pcd
lookup-pcd	*none*
extract-ptc	*none*

Table 8: Studies on JPM extension

attempted to enrich the pointcut designators because current AOP languages do not provide sufficient kinds of pointcut designators. G. Kiczales emphasizes the necessity of new kinds of pointcut designators such as `pcflow` (predictive control flow) and `dflow` (data flow)[16][20]. K. Gybels and J. Brichau have pointed out problems of current pointcut languages from the viewpoint of the software evolution, and have proposed robust pattern-based pointcut designators using logic programming facilities[11]. D.B. Tucker and S. Krishnamurthi propose a description of pointcuts and advice for higher-order languages, particularly Scheme[28]. These approaches introduce new pointcut designators in order to deal with new kinds of crosscutting concerns. These attempts to add new kinds of pointcut designators can be explained by the `register-one-pcd` protocol. We can introduce new pointcut designators such as `pcflow` by defining a pointcut evaluator. `Josh`[4], proposed by S. Chiba and K. Nakagawa, adopts an approach by which programmers can define a new pointcut designator as a boolean function. The approach of `Josh` can be considered as being similar to the `register-one-pcd` protocol. Although there exists research on pointcut extensions, research on extended join point types does not exist yet. This extension can be explained by `register-one-jp` protocols. For example, a rich join point such as `loop` can be defined by extracting points associated to control expressions such as `if`. Using the `register-one-strategy`, we can add new kinds of computation strategies. Although the current PA interpreter supports only the after-advice, we can define a new coordination (computation at a join point) for supporting the before-advice.

As mentioned in this section, the parameters of the X-ASB interpreter and the metaobject protocols derived from the parameters are useful not only for modeling different AOP mechanisms but also for discussing the research direction of extensible AOP.

6. RELATED WORK

There are two approaches for developing AOP languages that support multiple JPMs[12]. The first approach is to provide a single general-purpose AOP language that can design various special-purpose meta-level transformations. The second approach is to provide an AOP language with domain-specific aspect libraries.

This paper focuses on the first approach based on the three-part modeling framework. There are several works related to the first approach. E. Tanter et al. propose a versatile AOP kernel that supports core semantics[26]. J. Gray and S. Roychoudhury propose an approach that uses a program transformation system as the underlying engine for weaver construction[10]. Their long-term research goal, a framework for language and platform-independent weaving, is similar to our goal. R. Lämmel proposed a general method to extend the language in a way that it supports a form of superimposition just in the sense of AOP. In the extended language, a programmer can superimpose additional or alternative functionality (aka advice) onto points along the execution of a program[17]. The AspectBench Compiler (abc)[3] is a workbench that facilitates experimentation with new language features and implementation techniques. The abc is designed to be an extensible framework for implementing AspectJ extensions. X-ASB is an interpreter that can model not only AspectJ-like JPM such as PA but also other kinds of JPMs.

There are several AOP language systems adopting the second approach. M.Shonle, K.Lieberherr, and A.Shah propose an extensible domain-specific AOP language, XAspect, which adopts plug-in mechanisms[23]. Adding a new plug-in module, we can use a new kind of aspect-oriented facility. CME (Concern Manipulation Environment)[5], the successor of Hyper/J, adopts an approach similar to XAspect.

M. Mezini and K. Ostermann claim that join point interception (JPI) alone does not suffice for modular structuring of aspects[21]. They propose Caesar, a model of AOP with a higher-level module concept on top of JPI. Caesar enables reuse and componentization of aspects. There are two kinds of aspect components. One is a component that designs aspectual facilities as in the meaning of Caesar. Another is a component that designs mata-level transformations. Both of these components are important for extensible AOP. The second kind of aspect component establishes a foundation for new types of aspect-oriented features, and the first kind of aspect component gives a variety of aspectual features on the foundation. If we can write both kinds of components using the same base language, the above two approaches may be integrated. That is, the first approach (general-purpose AOP language with meta-level transformations) and the second approach (aspect library) corresponds to reflective AOP languages and associated reflective components, respectively.

Domain-specific aspect-oriented extensions are important. They are necessary not only at the programming stage but also at the modeling stage. An approach for supporting domain-specific aspect-oriented modeling is proposed in [9]. Logic programming facilities and queries using these facilities will be useful for defining domain-specific pointcuts[29]. If reflective AOP languages can expose program execution information held in weavers, and programmers can use this information when they define pointcuts, the pointcuts will be enriched.

7. CONCLUSION

This paper proposes the parameterized interpreter X-ASB for modeling different JPMs. X-ASB will be helpful in rapid-prototyping a new AOP mechanism or a reflective AOP system that supports multiple JPMs. We believe that X-ASB guides language developers in modular JPM designs.

8. ACKNOWLEDGEMENT

This research has been conducted under Kumiki Project, supported as a Grant-in-Aid for Scientific Research (13224087) by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan.

9. REFERENCES

[1] ASB(Aspect SandBox), http://www.cs.ubc.ca/labs/spl/projects/asb.html.

[2] AspectJ. http://www.eclipse.org/aspectj/.

[3] Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotak, J., Lhotak, O., Moor, O., Sereni, D., Sittampalam, G., and Tibble, J.: abc: An Extensible AspectJ Compiler, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pp.87-98, 2005.

[4] Chiba, S. and Nakagawa, K.: Josh: An Open AspectJ-like Language, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pp.102-111, 2004.

[5] Concern Manipulation Environment (CME): A Flexible, Extensible, Interoperable Environment for AOSD, http://www.research.ibm.com/cme/.

[6] Demeter Project. http://www.ccs.neu.edu/research/demeter/.

[7] Dutchyn, C., Kiczales, G., and Masuhara, H.: AOP Language Exploration Using the Aspect Sand Box, *Tutorial on International Conference on Aspect-Oriented Software Development (AOSD 2002)*, 2002.

[8] Elrad, T., Filman, R.E. and Bader A.: Aspect-oriented programming, *Communications of the ACM*, vol.44, no.10, pp.29-32, 2001.

[9] Gray, J., Bapty, T., Neema, S., Schmidt, D., Gokhale, A, and Natarajan, B.: An Approach for Supporting Aspect-Oriented Domain Modeling, In *Proceedings of International Conference on Generative Programming and Component Engineering (GPCE 2003)*, pp.151-168, 2003.

[10] Gray, J. and Roychoudhury, S.: A Technique for Constructing Aspect Weavers Using a Program Transformation Engine, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pp.36-45, 2004.

[11] Gybels, K. and Brichau, J.: Arranging Language Features for More Robust Pattern-based Crosscuts, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pp.60-69, 2003.

[12] Hugunin, J.: The Next Steps For Aspect-Oriented Programming Languages, http://www.isis.vanderbilt.edu/sdp, 2001.

[13] Kiczales, G., Rivieres, J.des , Bobrow, D. G.: The Art of the Metaobject Protocol, MIT Press, Cambridge, MA, 1991.

[14] Kiczales, G., Lamping, J., Mendhekar A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J.: Aspect-Oriented Programming, In *Proceeding of European Conference on Object-Oriented Programming (ECOOP'97)*, pp.220-242, 1997.

[15] Kiczales, G., Hilsdale, E., Hugunin, J., et al.: An Overview of AspectJ, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2001)*, pp.327-353, 2001.

[16] Kiczales, G.: The Fun Has Just Begun , *Keynote talk at International Conference on Aspect-Oriented Software Development (AOSD 2003)*, 2003.

[17] Lämmel", R.: Adding Superimposition To a Language Semantics, *Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2003 (FOAL 2003)*, 2003.

[18] Maes, P.: Concepts and Experiments in Computational Reflection, In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87)*, pp.147-155, 1987.

[19] Masuhara, H. and Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2003)*, pp.2-28, 2003.

[20] Masuhara, H. and Kawauchi, K.: Dataflow Pointcut in Aspect-Oriented Programming, In *Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03)*, pp.105-121, 2003.

[21] Mezini, M. and Ostermann, K.: Conquering Aspects with Caesar, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pp.90-99, 2003.

[22] Ossher, H. and Tarr, P.: Multi-Dimensional Separation of Concerns & Hyperspaces, *Software Architectures and Component Technology: The State of the Art in Research and Practice, Mehmet Aksit, editor*, Kluwer Academic Publishers, pp.293-323, 2000.

[23] Shonle, M., Lieberherr, K., and Shah, A.: XAspects: An Extensible System for Domain-specific Aspect Languages, *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003), Domain-Driven Development papers*, pp.28-37, 2003.

[24] Smith, B. C.: Reflection and Semantics in Lisp, In *Proceedings of Annual Symposium on Principles of Programming Languages (POPL'84)*, pp.23-35, 1984.

[25] Sullivan, G.T.: Aspect-oriented programming using reflection and metaobject protocols, *Communications of the ACM*, vol.44 no.10, pp.95-97, 2001.

[26] Tanter, E. and Noye, J.: A Versatile Kernel for Multi-Language AOP, In *Proceedings of Generative Programming and Component Engineering (GPCE 2005)*, to appear, 2005.

[27] Tarr, P., Ossher, H., Harrison, W. and Sutton, S.M., Jr.: N Degrees of Separation: Multi-dimensional Separation of Concerns, In *Proceedings of International Conference on Software Engineering (ICSE'99)*, pp.107-119, 1999.

[28] Tucker, D.B. and Krishnamurthi,S.: Pointcuts and advice in higher-order languages, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pp.158-167, 2003.

[29] Volder, K., Brichau, J., Mens, K., and D'Hondt, T.: Logic Meta Programming, a Framework for Domain-Specific Aspect Languages, http://www.cs.ubc.ca/ kdvolder/, 2001.

[30] Wand, M., Kiczales, G., and Dutchyn, C.: A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming, In *Proceedings: Foundations Of Aspect-Oriented Languages (FOAL2002), Workshop at AOSD 2002*, pp.1-8, 2002.

APPENDIX

Common

```
(define weave
  (lambda (pgm-a pgm-b)
    (register-jp)
    (eval-program pgm-a pgm-b)))
(define-struct jp (computation-strategy
     lookup-a effect-a lookup-b effect-b))
(define-struct pcd (pname evaluator))
(define computation-at-jp
  (lambda (jp)
    (let ((strategy (jp-computation-strategy jp))))
      (cond ((b-control-a? strategy)
             (computation-at-jp::b-control-a jp))
            ((relationship? strategy)
             (computation-at-jp::relationship jp))
            <other strategies are ommited>)))
(define computation-at-jp::b-control-a
  (lambda (jp)
    (let* ((lookup-a (jp-lookup-a jp))
           (effect-a (jp-effect-a jp))
           (lookup-b (jp-lookup-b jp))
           (effect-b (jp-effect-b jp)))
      (effect-b (lookup-b jp) jp
        (lambda ()
          (effect-a (lookup-a jp) jp))))))
(define computation-at-jp::relationship
  (lambda (jp)
    (let* ((lookup-a (jp-lookup-a jp))
           (effect-a (jp-effect-a jp)))
      (effect-a (lookup-a jp) jp))))
```

PA weaver

Body

```
(define pa::weave
  (lambda (pgm)
    (weave (extract-org-pgm pgm)
           (extract-advice-decls pgm))
    <extract-org-pgm extracts an original program>
    <extract-advice-decls extracts advice
     declarations>))
```

Join point type, pointcut designator

```
(define-struct (call-jp jp) (mname target args))
(define-struct (call-pcd pcd)
               (rtype cname mname params))
(define register-jp
  (lambda ()
    (register-one-jp
       'call-jp
       (lambda (mname target args)
         (make-call-jp
           'b-control-a
            lookup-method execute-method
            lookup-advice execute-advice
            mname target args)))
    (register-one-pcd
      'call-pcd
      (lambda (ptc jp)
        (if (call-pcd? ptc)
            (and (eq? (call-pcd-mname ptc)
                      (call-jp-mname jp))))))))
(define lookup-method
  (lambda(jp) <lookup a method>))
(define execute-method
  (lambda(method jp) <execute the method>))
(define lookup-advice
  (lambda (jp)
    <lookup advices that match pointcut conditions
      using pointcut evaluator 'call-pcd>))
(define execute-advice
  (lambda (advices jp thunk) <execute advices>))
```

Program evaluator

```
(define eval-program
  (lambda (org-pgm advice-decls)
    <evaluate expressions by calling eval-exp>))
(define eval-exp
  (lambda (exp env)
    (cond
      ((method-call-exp? exp)
       (call-method
         (method-call-exp-mname exp)
         (eval-exp
           (method-call-exp-obj-exp exp) env)
         (eval-rands
           (method-call-exp-rands exp) env)))
       <other expressions are ommitted>)))
(define call-method
  (lambda (mname obj args)
    (computation-at-jp
      (jtype-generator (lookup-jp 'call-jp))
       mname obj args)
    <lookup-jp searches a registered join point
     type>))
```

COMPOSITOR weaver

Body

```
(define compositor::weave
  (lambda (pgm-a pgm-b)
    (weave pgm-a pgm-b)))
```

Join point type

```
(define-struct (matching-jp jp)
               (pgm seed relationships))
(define register-jp
  (lambda ()
    (register-one-jp
       'matching-jp
       (lambda (pgm seed relationships)
         (make-matching-jp
          'relationship
           lookup-decl provide-decl
           lookup-decl provide-decl
           pgm seed relationships)))))
(define lookup-decl
  (lambda (jp)
    <lookup a declaration to be merged>))
(define provide-decl
  (lambda (decl jp)
    <add the declaration to a merged program>))
```

Program evaluator

```
(define eval-program
  (lambda (pgm-a pgm-b)
    (let loop
         ((pgm (make-program '()))
          (seeds (compute-seeds pgm-a pgm-b))
          (relationships (get-relationships))
            <compute seeds, all possible
             compositions of declarations>)
      <take a seed, and determine
         wheather the seed should be merged>
      <if so, generate a matching join point,
         and call computation-at-jp>
      (computation-at-jp
        (jtype-generator (lookup-jp 'matching-jp))
        pgm (car seeds) relationships)
      (loop pgm (cdr seeds) relationships))
     pgm))
```