ABSTRACT

There are several AOP (aspect-oriented programming) systems that deal with different kinds of crosscutting concerns. Moreover, there are crosscutting concerns specific to certain kinds of user applications, and these concerns may not be modularized with current AO mechanisms. Extensible AOP languages are needed to address this problem. In this paper, a framework for implementing extended AO mechanisms is proposed. The framework, called X-ASB (eXtensible Aspect Sand Box), is based on a common framework for modeling different AO mechanisms proposed by Masuhara and Kiczales. Using X-ASB, language developers can rapid-prototype new AO mechanisms for designing and evaluating new AOP languages. We believe that clarifying programming interfaces for extending AO mechanisms is an important milestone towards computational reflection for AOP.

1. INTRODUCTION

Mechanisms in AOP (aspect-oriented programming) [17][9] can be characterized by a JPM (join point model) consisting of the join points, a means of identifying the join points (pointcuts), and a means of raising effects at the join points (advice). JPMs are essential to AOP because they can deal with crosscutting concerns in general ways[34][33][35].

Crosscutting concerns may not be modularized as aspects without an appropriate join point definition that covers the interested elements in terms of the concerns, and a pointcut language that can declaratively identifies the interested elements. Each of current AOP languages is based on a few fixed set of JPMs. Many different JPMs have been proposed, and they are still evolving so that they could better modularize various crosscutting concerns.

We propose a common implementation framework, called X-ASB (eXtensible Aspect Sand Box), for AOP mechanisms in which different JPMs can be provided as its extension based on a modeling framework presented by Masuhara and Kiczales[22]. The modeling framework captures different JPMs by showing a set of interpreters. We call this the M&K model. X-ASB is based on contributions of ASB[2][8] that is a suite of aspect-oriented interpreters including PA (pointcuts and advice as in AspectJ[3][18]), TRAV (traversal specifications as in Demeter[6]), COMPOSITOR (class hierarchy composition as in Hyper/J[30][25]), OC (open classes as in AspectJ), and QB (query-based browsing as in JQuery[15]). Advantages of X-ASB are: language developers can easily prototype new or extended JPMs; more than one JPM can be provided at the same time like combining Demeter-like traversal mechanism in AspectJ-like advice. Most of current extensible AOP languages allows the programmers to extend the elements of the JPMs in their language, not to introduce new JPMs.

There are two approaches for developing extensible AOP languages[14]. The first approach is to provide a single general-purpose AOP language that can implement various special-purpose meta-level transformations. The second approach is to provide an AOP language with domain-specific aspect libraries. There have been no AOP languages so far that are based on the first approach because it has not been established as a language model that covers all kinds of JPMs. Without a language model, we cannot design metaobject protocols. On the other hand, language systems adopting the second approach, such as XAspect[26], have already been proposed. Aspect-oriented features can be enhanced by adding plug-in modules in these languages.

This paper focuses on the first approach based on the M&K model. Our final goal is computational reflection for AOP. We consider facilities of adding new JPMs or changing existing JPMs from base level languages as reflection for AOP. Although the M&K model clarifies common mechanisms in major JPMs, it does not provide a common implementation model that is needed to develop extensible AOP languages. In this paper, we describe our initial result for providing a common implementation model for JPMs, and clarifying programming interfaces for introducing or extending JPMs. We believe that the result is an important milestone towards computational reflection for AOP.

In this paper, issues on implementing AOP languages are pointed out in section 2. In section 3, the M&K model is briefly explained. Subsequently, in section 4, X-ASB is introduced in order to tackle issues pointed out in section 2. We demonstrate how easily and modularly JPMs can be
implemented using X-ASB in section 5. In section 6, we show advanced topics towards computational reflection for AOP. In section 7, the usefulness of X-ASB is evaluated in terms of the efficiency and the easiness of introducing new kinds of JPMs. In section 8, we introduce some related work on extensible AOP languages, and discuss future directions of research. Lastly, in section 9, we conclude this paper.

2. ISSUES ON IMPLEMENTING AOP LANGUAGES
Designing and implementing a new language is not easy. Although extensible languages, such as computational reflection[27][20] and metaobject protocols[16] would be useful, providing an extensible AOP language that covers possible JPMs is not easy because JPMs are drastically different from the viewpoint of implementation. Without a common implementation model, we cannot realize computational reflection.

It is very difficult to find the commonality among implementations of current major AOP languages. The difference of implementation is caused by a wide variety of JPMs that AOP languages support. For example, join points in AspectJ are defined explicitly. Join points in Demeter, on the other hand, are defined implicitly by traverse strategies. We will examine these differences using two brief sample programs written in AspectJ and Demeter[3][6].

AspectJ (PA)
provides join points as certain points in the execution flow of the program. For example, method call join point is the point in the flow when a method is called. Pointcut designators identify particular points from all the join points in the flow. The following AspectJ program defines an aspect consisting of a pointcut move and an after advice. The pointcut catches the execution points of method call that moves figure elements, and inserts code for display updating at the points. The PA weaver in ASB creates an AST (Abstract Syntax Tree), walks the AST, and executes advice at join points. The PA weaver in ASB creates an AST, an object graph in TRAV, and so forth; and

```
aspect DisplayUpdating {
  pointcut move(FigureElement figElt): 
    target(figElt) &&
    (call(void FigureElement.moveBy(int, int)) ||
     call(void Line.set*(Point)) ||
     call(void Point.set*(int)))
  after(FigureElement fe) returning: move(fe) {
    Display.update(fe);
  }
}
```

Demeter (TRAV)
provides a mechanisms that enables programmers to implement traversals through object graphs in a succinct fashion. The following program is a salary calculation program that traverses the departments of a company. The traverse strategy is “from Company to Salary”. The visitor traverses an object graph, and executes the visitor method before. Lastly, the method getReturnValue is executed. An arrival at a node of an object graph is a join point, and an execution of a visitor method is an advice. In the adaptive programming that Demeter supports, a program need not be changed even if new objects such as new department objects are added to an object graph. The TRAV weaver in ASB creates an object graph, walks the graph, and executes visitor method at each node.

```
class Company {
  // class structure
  static ClassGraph cg = new ClassGraph();
  Double sumSalaries() {
    // traversal strategy
    String s = "from Company to Salary";
    // adaptive visitor
    Visitor v = new Visitor() {
      private double sum;
      public void start() {
        sum = 0.0;
      }
      public void before(Salary host) {
        sum += host.getValue();
      }
      public Object getReturnValue() {
        return new Double(sum);
      }
    };
    return (Double)cg.traverse(this, s, v);
  }
  // ... rest of Company definition ...
}
```

3. MASUHARA & KICZALES MODEL
Although several differences among JPMs may make it difficult to actually implement a single parameterizable procedure, we believe that there is some kind of implementation structures that can be commonly applied to major JPMs. Indeed, we can observe an abstracted common weaving process as follows:

1. parse source code, and create an internal data structure including an AST in PA, an object graph in TRAV, and so forth; and
2. analyze the internal data structure, create a join point, check if the join point satisfies conditions specified by pointcut designators, and execute advice if these conditions are true.

The M&K model[22] shows a common model that can be used to explain how PA, TRAV, COMPOSITOR, OC, and QB support modular crosscutting. The model shows the core semantics of these mechanisms by modeling the weaving process by which they are implemented. The model defines the process of weaving as taking two programs and coordinating them into a single combined computation. A critical property of the model is that it describes the join points as existing in the result of the weaving process rather than being in either of the input programs

The M&K model explains each aspect-oriented mechanism as a weaver that is modeled as a tuple of 9 parameters:

(X, XJP, A, AID, AEFF, B, BID, BEFF, META).

A and B are the languages in which the respective programs pA and pB input to the weaver are written. X is the result domain of the weaving process, which is the third language...
of a computation. \( X_{jp} \) is the 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 effecting semantics at the identified join points. \( META \) is an optional meta-language for 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.

In terms of the M\&K model, PA is modeled as follows: \( X \); execution of combined programs; \( X_{jp} \); method calls, field gets, and field sets; \( A \); class, method, and field declarations; \( A_{id} \); method and field signatures; \( A_{eff} \); execute method body; \( B \); advice declarations with pointcuts; \( B_{id} \); pointcuts; \( B_{eff} \); execute advice body before, after, and around method. TRAV, on the other hand, is modeled as follows: \( X \); execution of a traversal through an object graph (visit objects specified by the traversal specification); \( X_{jp} \); arrival at each object along the traversal; \( A \); class and field declarations; \( A_{id} \); class names and complete field signatures; \( A_{eff} \); reachability information; \( B \); traversal specification and visitor; \( B_{id} \); traversal specification, overloaded visitor methods; \( B_{eff} \); call visitor and continue traversal (or not).

4. X-ASB: A FRAMEWORK FOR EXTENDING JPMS

In this section, we propose X-ASB as one of the common implementation structures for major JPMS. Although the M\&K model parameterizes major JPMS, it makes no mention of implementation structures. In the current ASB implementation based on the M\&K model, weavers are developed individually, and there is no common implementation among these weavers. Moreover, it is impossible to add new kinds of join points and pointcut designators unless the code of each weaver is re-implemented, and code regions to be modified are scattered.

4.1 Framework layer

There are multiple framework layers for implementing or extending JPMS as shown in Figure 1. The level 1 framework provides the common implementation for all kinds of JPMS and programming interfaces that must be implemented by JPM developers. The interfaces expose hot-spots for extending JPMS. This level gives JPM developers basic architecture for implementing JPMS. The level 2 framework provides advanced toolkit for implementing specific weavers such as PA-like weavers and TRAV-like weavers. Using the toolkit, JPM developers can implement individual weavers as well as multi-paradigm weavers that support several JPMS. For example, COMPOSITOR that supports PA-like before/after advice can be implemented. In this section, we show the overview of the level 1 framework that is currently provided by X-ASB. The framework based on the M\&K model adopts nine parameters including \( X \), \( X_{jp} \), \( A \), \( A_{id} \), \( A_{eff} \), \( B \), \( B_{id} \), \( B_{eff} \), and \( META \) as hot-spots for extending JPMS.

4.2 X-ASB overview

The overview of X-ASB, which is implemented in Scheme, is shown in Figure 2. The bottom half is a common implementation provided by X-ASB, and the top half is a set of programming protocol interfaces that must be implemented by JPM developers. The common implementation includes a base language interpreter, libraries provided for JPM developers, and other common implementations that are not shown here. Table 1 shows programming protocols as function names with their parameter name lists. Using the interfaces, a new kind of JPM can be added to the base language. The interfaces expose hot-spots for defining and registering join points (no.2), pointcuts (no.3), and advice (no.4, 5, 6, 7) because a JPM is characterized by these three components. The interfaces also expose hot-spots for defining a weaver body that mediates these components (no.1). Table 2 shows X-ASB libraries.

In X-ASB, JPMS can be systematically introduced or extended as follows:

1. define kinds of join points (define register-\( \text{jp} \));
2. define kinds of pointcuts (define register-\( \text{pcd} \)); and
3. define a body of weaver, and computation at join points (define eval-program, etc.).
The base language interpreter calls the functions `register-jp`, `register-pcd`, and `eval-program` implemented by JPM developers as follows:

```
(define weaver
  (lambda (pgm meta)
    (register-jp)
    (register-pcd)
    (eval-program pgm meta)))
```

We show a JPM development process using the code skeleton of the PA weaver that have only method `call` join point and call pointcut designator.

**Step 1: define kinds of join points.**
First, JPM developers have to define kinds of join points and the related data structures including an AST in PA, an object graph in TRAV, and so on. The interface `register-jp` and its parameter `generator` are used in the step 1 (see 1, 1-1, 1-2 in Figure 2). The `register-jp` interface registers a new kind of join point. Each join point is managed by the structure composed of a join point tag name and a `generator` that generates a join point. In the base language interpreter, there are several hook-points such as `call-method`, `var-set/get`, `field-set/get`, `if`, and so forth. A set of join points can be selected from these hook points. Crosscutting concerns such as loop structures can be extracted by selecting hook points concerning control expressions as join points. Crosscutting concerns such as data flows, on the other hand, can be extracted by selecting data access hook points. In general, data structures related to join points tend to be different drastically among JPMs. The `generator` parameter abstracts differences among these data structures. The `register-one-jp` library function helps JPM developers to implement the `register-jp` interface. The following is the code skeleton of the PA weaver.

```
(define register-jp
  (lambda ()
    (register-one-jp 'call-method generate-call-jp)))
(define generate-call-jp ...)
```

**Step 2: define kinds of pointcuts.**
Next, kinds of pointcut designators must be defined as a boolean function using the `register-pcd` interface (see 2, 2-1, 2-2 in Figure 2). Each pointcut designator is managed by the structure composed of a pointcut tag name and an `evaluator` that checks whether a current join point is an element of a pointcut set. The `register-one-pcd` library function helps JPM developers to implement the `register-pcd` interface. The following is the code skeleton of the PA weaver.

```
(define register-pcd
  (lambda ()
    (register-one-pcd 'call call-pcd?)))
(define call-pcd? ...)
```
Step 3: define a body of weaver, and computation at join points.

Lastly, JPM developers have to implement a body of a weaver using the eval-program interface that corresponds to the X parameter in the M&K model (see 3 in Figure 2). In the case of the PA weaver, the internal data structure related to join points is an AST. The eval-program creates an AST, walks it, and checks if the visited node is registered as a join point. At the method call join point, the call-method function related to the AST is called. Figure 3 illustrates the architecture of the PA weaver. The following is the body of the PA weaver.

```scheme
(define eval-program
  (lambda (pgm meta)
    (walk-ast ; generate & walk AST
      (generate-ast pgm meta)))))

(define walk-ast
  (lambda (ast)
    ... ; call-method mname obj args ; computation at method call ...
    ; join point

    (call-method mname obj args) ; computation at method call ...
    ...))

The computation at the method call join point, the call-method function, can be defined using the computation-at-jp library function, a generic (template) function as shown below (see 5, 6 in Figure 2)\(^1\). The jp parameter (see 4-3 in Figure 2) is an instance of a current join point generated by the generator (see 4-2 in Figure 2) that is registered by register-jp (see 1-2 in Figure 2). The registered join point generator can be searched using the lookup-jp-generator library function (see 4, 4-1 in Figure 2). The lookup-A-ID/lookup-B-ID and effect-A/effect-B interfaces correspond to \(A_{ID}/B_{ID}\) and \(A_{EFF}/B_{EFF}\) in the M&K model, respectively. These interfaces must be implemented by JPM developers. Implementing the interfaces, a new kind of advice can be introduced. In call-method, the call pointcut evaluator call-pcd? is executed in the pointcut-matches (see 8 in Figure 2) invoked from the lookup-B-ID (see 7 in Figure 2), and the advice executor effect-B is executed.

```scheme
;; X-ASB library
(define computation-at-jp
  (lambda (jp param)
    (let* ((A-ID (lookup-A-ID jp param))
           (B-ID (lookup-B-ID jp param)))
      (effect-B B-ID jp
        (lambda ()
          (effect-A A-ID jp param)) param))))

(define pointcut-matches
  ... search a pointcut evaluators corresponding to a join point, and execute a found evaluator.)

;; define computation at call method join point
;; using X-ASB library
(define call-method
  (lambda (mname obj args)
    (computation-at-jp
      ((lookup-jp-generator 'call-method) mname obj args) null))) ; no additional parameter

(define lookup-A-ID ...) ; lookup method
(define lookup-B-ID ...) ; lookup advice
(define call-pcd? ...) ; check if the join point satisfies the pointcut conditions
(define pointcut-matches ... search a pointcut evaluators corresponding to a join point, and execute a found evaluator.)
(define effect-A ...) ; execute method
(define effect-B ...) ; execute-advice
```

As shown in step 1, 2, and 3, the PA weaver is constructed modularly using X-ASB. JPM developers have only to modify specific code regions such as register-jp and register-pcd when a new kind of join point and pointcut are needed. On the other hand, JPM developers must modify several code regions in order to add a new JPM element in the case of the current ASB implementation. We can separate JPM implementations using X-ASB. Separation of implementation concerns contributes to evolution of JPMs.

\(^1\)In the computation-at-jp, the order of execution of effect-A and effect-B are hard-coded in the framework. That is, effect-B is executed after effect-A. There is no problem when the framework is applied to development of the PA-like weaver in which aspects dominate classes. However, the above code cannot be applied when we develop a symmetric JPM such as COMPOSITOR. Moreover, a specific policy of effecting at a join point may be needed. It is necessary to provide variation of the computation-at-jp in the framework.
5. IMPLEMENTING WEAVERS USING X-ASB

In this section, we demonstrate how easily and modularly other JPMs can be implemented using X-ASB. Two implementations including the TRAV weaver and an extended PA weaver, are shown.

5.1 TRAV weaver

In TRAV, internal data structure related to join points can be indirectly given as a path of an object graph that a visitor traverses, and the pointcut evaluator checks whether an object node on the path satisfies the traversal specifications. The eval-program creates the object graph, and walks it. The pgm parameter in the eval-program is indirectly given as a root node of an object graph that indicates a program structure, and the meta parameter is a list composed of the traversal specification and visitor. A computation of each join point is given as an execution of a visitor method at a node on the path. Thus, two interfaces lookup-A-ID and effect-A are used implicitly. On the other hand, the lookup-B-ID interface and the effect-B interface are used explicitly. They correspond to the function for looking up a visitor and the function for executing the visitor method, respectively.

(define eval-program
  (lambda (pgm meta)
    (let ((root pgm)
          (trav-spec (car meta))
          (visitor (cadr meta)))
      (let arrive ((obj root)
                   (path (make-path (object-cname root))))
        (execute-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 (pointcut-matches trav-spec next-path)
                          (arrive next-obj next-path))))
       (object->fnames obj)))))))

(define execute-call-visitor
  (lambda (visitor obj)
    (computation-at-jp ((lookup-jp-generator 'trav) obj)
                        visitor)))

;; register-jp
(define register-jp
  (lambda () (register-one-jp 'trav generate-trav-jp)))
(define generate-trav-jp (lambda (obj) obj))

;; register-pcd
(define register-pcd
  (lambda () (register-one-pcd 'trav generate-trav-pcd?)))
(define trav-pcd? ...
  ... ; check whether a next object
  ... ; path satisfies the traversal
  ... ; specifications)

;; lookup-A-ID, lookup-B-ID, effect-A, effect-B
(define lookup-A-ID ...) ; dummy
(define lookup-B-ID ...) ; lookup visitor
(define effect-A ...) ; dummy
(define effect-B ...) ; execute visitor method

Figure 4 illustrates the architecture of the TRAV weaver. We can develop both of PA and TRAV in the same fashion using the X-ASB framework and programming interfaces.

5.2 Extended PA weaver

It is desirable that one can extend an existing weaver slightly when JPMs that the weaver provides are insufficient for describing the target problem. It is relatively easy to deal with this problem using X-ASB. The following is an example in which the PA weaver is extended in order to support context-sensitive calling sequences. The example is a communication program in which a protocol —an order of exchanged messages— is important. This program, written in the base language of X-ASB, separates a situation in which a protocol error might occur by defining a new kind of pointcut designator. Suppose that the order of message sequences is <m1, m2, m3>. The pointcut definition (calling-sequence (not (list 'm1 'm2 'm3))) catches the crosscutting concern that violates the order.

(class sample-protocol-error-detection object
  (method int m1 () (...))
  (method int m2 () (...))
  (method int m2 () (...))
  (after (calling-sequence (not (list 'm1 'm2 'm3)))
             (write 'invalid-calling-sequence) (newline)))

This pointcut designator can be added to the existing PA weaver as follows.
6. TOWARDS REFLECTION

The X-ASB exposes two kinds of programming interfaces for adding JPMs to the base language. The first is a set of programming interfaces provided for JPM developers that implement weavers in Scheme. Using these programming interfaces, a concrete weaver including the PA weaver and the TRAV weaver can be developed as mentioned in section 5. In this case, JPM developers are separated from application programmers. The second is a set of programming interfaces provided for programmers that develop user applications and want to add JPMs specific to these applications. These programming interfaces are exposed for application programmers to use the facilities of the first kind of programming interfaces within the base language. In this case, JPM developers are the same as application programmers. In the implementation style illustrated in subsection 5.2, only JPM developers can extend the PA weaver.

It would be better for application programmers to be able to add new aspect-oriented features using X-ASB programming interfaces within the base language, as follows. Application programmers may override the register-pcd method defined in the object class. To call functions defined in the framework provided by X-ASB, programmers may use the meta call.

```
(define register-pcd
  (lambda ()
    (register-one-pcd 'calling-sequence
    calling-sequence-pcd?)))
(define calling-sequence-pcd? ...)  
```

Although the power of the extension is still limited, this brings to mind the reflective programming. The base language programming interfaces in X-ASB correspond to metainterface protocols in reflective OOP languages. In the above program, the meta call to the register-one-pcd interface and an associated method calling-sequence-pcd? can be encapsulated as a component for extending JPMs. It is important that the component is written in the same base language. We can build user libraries for extending JPMs by accumulating this kind of components. We believe that reflective languages and associated reflective components will be very important for realizing domain-specific AOP in the future. If we can obtain this goal, two approaches for developing extensible AOP languages, remarked in section 1, will 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.

There is a big problem to realize computational reflection for AOP. Although JPMs such as COMPOSITOR need compile-time weaving, JPMs such as TRAV need run-time weaving. PA can be implemented by either of them. We must integrate compile-time weaving and run-time weaving into a single interpreter. We propose the idea of multi-stage weaving to address this issue. Major JPMs have the same implementation architecture except internal data structures related to join points. We demonstrates the fact in section 4 and 5. In the multi-stage weaving, each stage in compile-time weaving translates source code to an internal data structure, analyzes the data structure, creates a join point, checks the join point using a pointcut evaluator, transforms the data structure using advice executor, and lastly translates the transformed data structure to source code. Figure 5 illustrates this process. Each stage can be realized as a graph transformation because internal data structures can be represented as graph structures such as AST in many cases[1]. In a run-time weaving stage, each JPM is applied by specifying a scope in which the same process in compile-time weaving is executed. Each stage in the multi-stage weaving exposes programming interfaces introduced in the section 4 as hot-spots for computational reflection. Introducing the idea of multi-stage weaving, several kinds of JPMs can exist in one source code. We are developing R-ASB (Reflective ASB) that realizes computational reflection for AOP based on the multi-stage weaving now.

7. EVALUATION

In this section, we evaluate the effectiveness of X-ASB from the following viewpoints.
efficiency of developing a new JPM
• easiness of developing a new JPM
• performance of a developed weaver

7.1 Development efficiency
Table 3 shows the code size of the base interpreter, the framework, developed weavers based on the framework, and original ASB weavers. Although the size of the original PA weaver is 189 LOC (Line of code), the size of the X-ASB framework and the extended PA weaver is 241 LOC. The latter size becomes larger than the former size. However, the size of the code that programmer must write is only 171 LOC, and the effort for developing the PA weaver decreases 9.5%. The additional code for the framework adaptation is 45 LOC, and 126 LOC can be reused from the original PA weaver. The code size 63(=189-126) LOC is shifted from the original PA weaver to the framework as a common implementation. On the other hand, the advantage of the adoption for developing TRAV is relatively small because TRAV needs a specific join point data structure (an object graph), and cannot always reuse the common implementation in the framework. Although the advantage of applying X-ASB to the TRAV weaver is little in terms of the code size, X-ASB gives us architecture for building weavers.

7.2 Development easiness
We evaluated the development easiness in terms of the modularity in adding new kinds of JPMs, especially join points and pointcut designators. A new join point is defined by a new kind of data structure, and a new pointcut designator is defined by a boolean function that checks if a join point satisfies the pointcut conditions. Table 4 shows the size of the code for adding the call-method join point and the call pointcut. The total necessary code size is 71 LOC, and this implementation is encapsulated in the code region for the framework adaptation. The code size of the PA weaver based on X-ASB is 241 LOC as shown in Table 3, and the percentage of the modified code is 29%. This is the biggest advantage of using X-ASB because we can extend JPMs modularly. That is, hot spots for adding new kinds of join points and pointcut designators are better modularized in comparison with the original ASB implementation. In ASB, the code for adding them is scattered in several code regions. There is another advantage in using X-ASB. As shown in section 4, programmers can implement new weavers easily because the guideline for adding new kinds of JPMs is provided in X-ASB.

7.3 Performance
In X-ASB, JPMs consisting of join points, pointcut designators, and advice are not hard-coded, but registered to an internal list using programming interfaces. We carried out a benchmark test to investigate how the searching speed of these registered elements affects the performance. The benchmark test was executed by DrScheme[7] profiler, running on a Pentium 4 2.6GHz Windows XP Professional machine with 512MB memory. Figure 6 shows execution times of the pointcut-matches in which pointcut evaluators are searched. The horizontal axis is number of pointcut evaluators. We used a test program that have ten join points. Each pointcut evaluator is executed in the pointcut-matches whenever weaver encounters each join point. We used the function call-pcd? and functions that only return #f as pointcut evaluators. In the test, the call-pcd-check? was lastly checked. The execution times increase linearly as the number of pointcut designators increases. However, we do not consider the result as a big problem because there are not so many kinds of pointcut designators in practice.

8. RELATED WORK
Shonle, Lieberherr, and Shah propose an extensible domain-specific AOP language XAspect that adopts plug-in mechanisms[26]. 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. Although the concept of extensible AOP languages is very useful, there will be cases in which a new AOP language suited to specific problems must be developed[14]. Domain-specific aspect-oriented extensions are necessary not only in programming stages but also in modeling stages. An approach for supporting aspect-oriented domain modeling is proposed in [11].

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC (Line of code)</th>
</tr>
</thead>
<tbody>
<tr>
<td>base interpreter framework</td>
<td>1183</td>
</tr>
<tr>
<td>weaver (X-ASB)</td>
<td>70</td>
</tr>
<tr>
<td>weaver (X-ASB)</td>
<td>171(45), 230(56)</td>
</tr>
<tr>
<td>adaptation code: 45(126), 56(174)</td>
<td></td>
</tr>
<tr>
<td>reused code: 126(189), 174(174)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Code size for developing new weavers

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>code for registering join point</td>
<td>12</td>
</tr>
<tr>
<td>code for pointcut construct</td>
<td>15</td>
</tr>
<tr>
<td>code for computation at join point</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 4: Code size for adding call-method join point and call pointcut

![Figure 6: Execution times (in msec.) of pointcut-matches](image)
Although pointcut languages play important roles in AOP paradigms, current AOP languages do not provide sufficient kinds of pointcut designators. In an effort to address this problem, several previous investigations have attempted to enrich the pointcut designators. Kiczales emphasizes the necessity of new kinds of pointcut designators such as \texttt{pcflow} (predictive control flow) and \texttt{dflow} (data flow)\cite{19}. \texttt{Dflow} is implemented by Masuhara and Kawauchi\cite{23}. Gybels and Brichau point out problems of current pointcut languages from the viewpoint of the software evolution, and propose robust pattern-based pointcut designators using logic programming facilities\cite{13}. These approaches introduce new pointcut designators in order to deal with new kinds of cross-cutting concerns. However, adopting these approaches, we need to add another pointcut designator to existing AOP languages whenever we face another kind of problem. As a consequence, the syntax of AOP languages would become very complex. Chiba and Nakagawa propose \texttt{Josh}\cite{4} in which programmers can define a new pointcut designator as a boolean function. Using X-ASB, we can add not only new pointcut designators but also new kinds of join points and advice. Gray and Roychoudhury propose an approach that uses a program transformation system as the underlying engine for weaver construction\cite{12}. Their long-term research goal, a framework for language and platform-independent weaving, is similar to our goal.

In general, problems that contain particular kinds of cross-cutting concerns require particular kinds of join points and pointcut & advice constructs fit to the problem structure. When programmers make a model of target problems, they need simultaneously to design aspect-oriented features specific to the problems. To support programmers, it is necessary to provide reflective AOP languages. The example in section 6 implies this necessity. However, reflective AOP languages might not provide sufficient efficiency. This problem must be resolved before these languages are available for practical application. In addressing these problems, partial evaluations will be helpful\cite{21}. Logic programming facilities and queries using these facilities will be useful for defining domain-specific pointcuts\cite{32}. 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. We believe that research on the combination of reflection and query-based logic programming will be very important in the future. Most of aspect-oriented features can be implemented by a run-time metaobject protocols\cite{28}. It is interesting to explore relationships between reflective OOP languages and reflective AOP languages. We think that reflective AOP languages should expose programming interfaces based on JPM because it is essential to AOP. Tucker and Krishnamurthi propose a description of pointcuts and advice for higher-order languages, particularly Scheme\cite{31}. This research is interesting for us because X-ASB is also written in Scheme.

Mezini and Ostermann claim that join point interception (JPI) alone does not suffice for a modular structuring of aspects\cite{24}. They propose Caesar, a model of AOP with higher-level module concept on top of JPI, which enables reuse and componentization of aspects. There are two kinds of aspect component. One is a component that implements aspectual facilities as in the meaning of Caesar. Another is a components that implements meta-level transformations as shown in section 6. Both of these components are important for domain-specific AOP. The second kind of aspect components gives foundations for the new type of aspect-oriented features, and the first kind of aspect components gives a variety of aspectual features on the foundations.

9. CONCLUSION

This proposed X-ASB, a framework for introducing or extending JPMs. The framework exposes programming interfaces for adding new kinds of join points and pointcut & advice constructs. Using X-ASB, language developers can rapid-prototype new JPMs. We believe that clarifying programming interfaces for extending JPMs is an important milestone towards computational reflection for AOP.

10. REFERENCES


