An AOP Implementation Framework for Extending Join Point Models

Naoyasu Ubayashi
Kyushu Institute of Technology, Japan
ubayashi@acm.org

Hidehiko Masuhara
University of Tokyo, Japan
masuhara@acm.org

Tetsuo Tamai
University of Tokyo, Japan
tamai@acm.org

ABSTRACT

Mechanisms in AOP (aspect-oriented programming) can be characterized by a JPM (join point model). AOP is effective in unanticipated software evolution because crosscutting concerns can be added or removed without making invasive modifications on original programs. However, the effectiveness would be restricted if new crosscutting concerns cannot be described with existing JPMs. Mechanisms for extending JPMs are needed in order to address the problem. In this paper, an implementation framework for extending JPMs is proposed. Using the framework, we can introduce new kinds of JPMs or extend existing JPMs in the process of software evolution.

1. INTRODUCTION

Mechanisms in AOP (aspect-oriented programming) 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). 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.

AOP is effective in unanticipated software evolution because crosscutting concerns can be added or removed without making invasive modifications on original programs. However, the effectiveness would be restricted if new crosscutting concerns cannot be described with JPMs supported by current AOP languages. Mechanisms for extending JPMs are needed in order to address the problem. Masuhara and Kiczales presented a modeling framework that captures different JPMs by showing a set of interpreters[7]. We call this the M&K model. Based on the M&K model, we propose a framework, called X-ASB (eXtensible Aspect Sand Box), for implementing extensible AOP languages in which different JPMs can be provided as its extension. The term *framework* in this paper indicates provision of common implementations and exposure of programming interfaces for extending base languages. X-ASB is based on contributions of ASB[1] that is a suite of aspect-oriented interpreters such as PA (pointcuts and advice as in AspectJ) , TRAV (traversal specifications as in Demeter), and COMPOSITOR (class hierarchy composition as in Hyper/J). Advantages of X-ASB are: language developers can easily prototype new or extended JPMs in the process of software evolution; 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. 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. The effectiveness in software evolution would be restricted if language developers must extend JPMs whenever application programmers need new kinds of JPMs. Reflective mechanisms will address this problem.

In this paper, issues on implementing AOP languages are pointed out in section 2. In section 3, X-ASB is introduced to tackle the issues. We show a JPM development process from the viewpoint of software evolution. In section 4, we show advanced topics towards computational reflection for AOP. In section 5, we discuss the effectiveness of X-ASB in software evolution. We introduce some related work in section 6, and conclude the paper in section 7.

2. ISSUES ON IMPLEMENTING AOP LANGUAGES

Designing and implementing a new language is not easy. Although extensible languages, such as computational reflection[6] and metaobject protocols would be useful for software evolution, providing an extensible AOP language that covers possible JPMs is not easy because the JPMs are drastically different from the viewpoint of implementation.

The M&K model shows the semantics of major JPMs by modeling 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:

$$< X, X_{JP}, A, A_{ID}, A_{EFF}, B, B_{ID}, B_{EFF}, META > .$$

A and B are the languages in which the programs p_A and p_B 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}

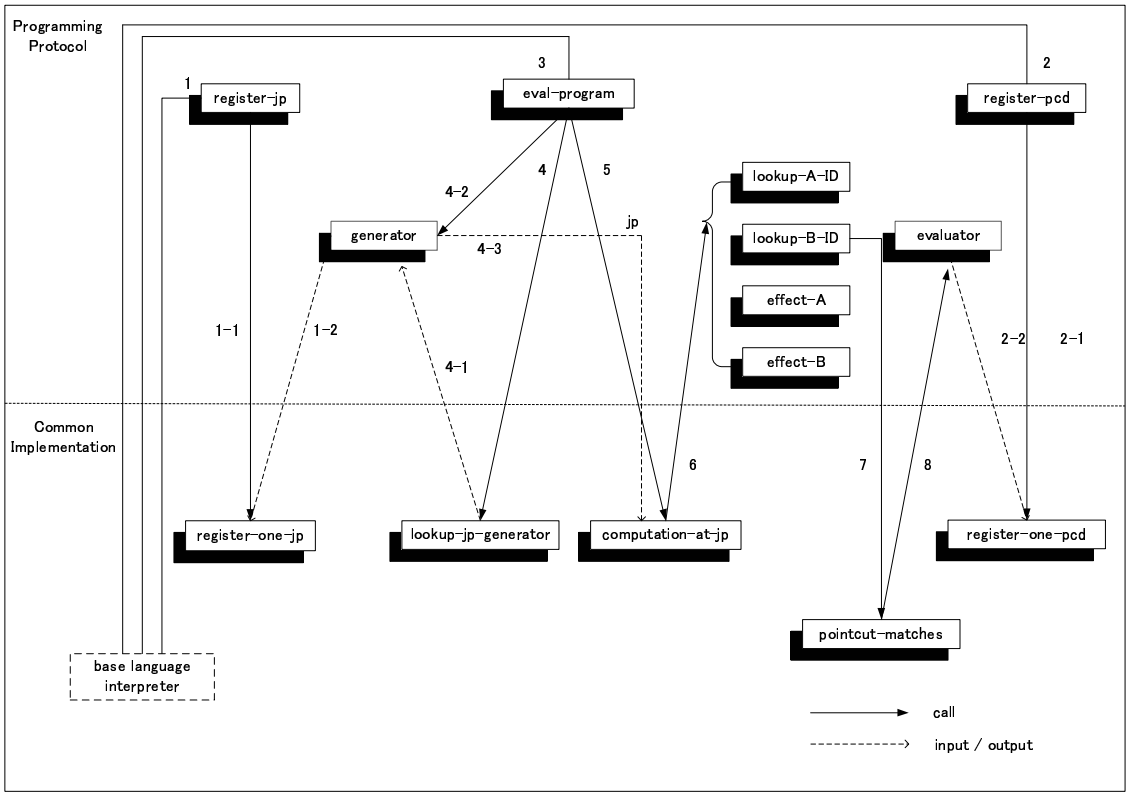

Figure 1: X-ASB overview

and B_{EFF} are the means 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; B: advice declarations with pointcuts; B_{ID}: pointcuts; B_{EFF}: execute advice before, after, and around method.

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. It is impossible to add new kinds of JPMs unless the code of each weaver is re-implemented, and code regions to be modified are scattered. 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. In the next section, we propose X-ASB as one of the common implementation structures.

3. X-ASB: A FRAMEWORK FOR EXTENDING JPMS

There are multiple framework layers for implementing or extending JPMs. 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 explain the overview of the level 1 framework that is currently provided by X-ASB. We show a JPM development process using the code skeleton of the PA weaver and the extended PA weaver from the viewpoint of software evolution.

3.1 X-ASB overview

The overview of X-ASB, which is implemented in Scheme, is shown in Figure 1. 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

No.	Signature	Ret val	M&K
1.	`(eval-program pgm meta)`	none	X
2.	`(register-jp tag generator)`	none	X_{JP}
3.	`(register-pcd tag evaluator)`	none	X_{JP}
4.	`(lookup-A-ID jp param)`	A_{ID}	A_{ID}
5.	`(lookup-B-ID jp param)`	B_{ID}	B_{ID}
6.	`(effect-A A-ID jp param)`	none	A_{EFF}
7.	`(effect-B B-ID jp thunk param)`	none	B_{EFF}

Table 1: X-ASB programming protocol

No.	Signature	Ret val
1.	`(register-one-jp tag generator)`	none
2.	`(lookup-jp-generator tag)`	generator
3.	`(register-one-pcd tag evaluator)`	none
4.	`(pointcut-matches ptc jp)`	#t or #f
5.	`(computation-at-jp jp param)` `param: optional information`	none

Table 2: X-ASB library

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; 2) define kinds of pointcuts; and 3) define a body of weaver, and computation at join points. 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. The PA weaver processes, for example, the following program that calculates the factorial of n. Calls to procedure `fact` is declared as a pointcut, and `after` advice is executed at the join point corresponding to the pointcut.

```
(class sample-fact object
  (method void init () (super init))
  (method int main () (send this fact 6)) ;call method
  (method int fact ((int n))
    (if (< n 1) 1
        (* n (send this fact (- n 1)))))
   ;pointcut & advice
  (after (call fact) (write 'after) (newline)))
```

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 (Abstract Syntax Tree) 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 1). 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 1). 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?  ...)
     ; compare method name of pointcut designator
     ; and method name of method call join point
```

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 `exec-program` interface that corresponds to the X parameter in the M&K model (see 3 in Figure 1). 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 2 illustrates the architecture of the PA weaver. The following is the body of the PA weaver.

```
(define eval-program
  (lambda (pgm meta)
    (walk-ast (generate-ast pgm meta))))
(define walk-ast
  (lambda (ast)
     ...
    ; computation at method call join point
    (call-method mname obj args)
     ...))
```

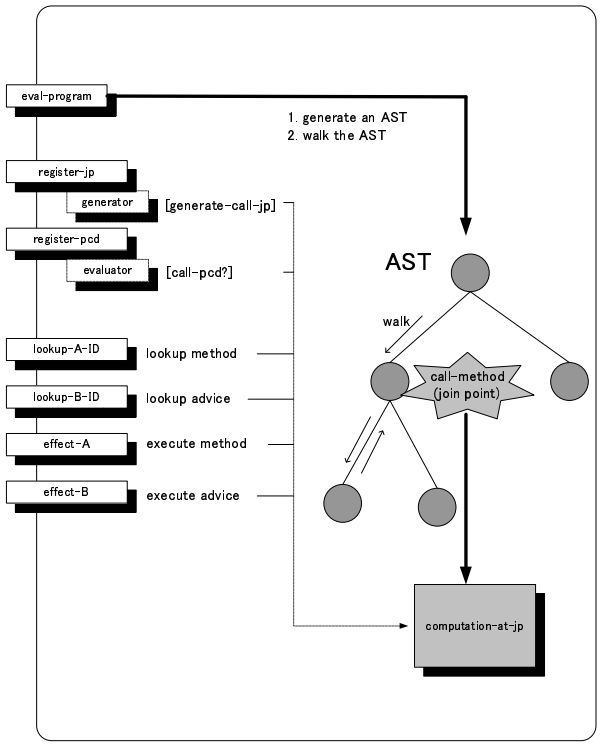

Figure 2: PA weaver overview

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 1). The `jp` parameter (see 4-3 in Figure 1) is an instance of a current join point generated by the generator (see 4-2 in Figure 1) that is registered by `register-jp` (see 1-2 in Figure 1). The registered join point generator can be searched using the `lookup-jp-generator` library function (see 4, 4-1 in Figure 1). The `lookup-A-ID`/`lookup-B-ID` and `effect-A`/`effect-B` interfaces corresponds 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 1) invoked from the `lookup-B-ID` (see 7 in Figure 1), and the advice executor `effect-B` is executed.

```
;; 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
                              ; (check if the join point
                              ;  satisfies the pointcut
                              ;  conditions
                              ;  using pointcut-matches.
                              ;  return advice if true.)
(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.

3.2 Extending existing weaver

It is desirable that one can extend an existing weaver slightly when JPMs that the weaver provides are insufficient for describing new kinds of features required in the process of software evolution. 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 construct. 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.

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

4. TOWARDS REFLECTION

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 language developers that implement weavers as shown in section 3. The second is a set of programming interfaces provided for programmers that develop user applications and want to add JPMs specific to these applications. In the implementation style illustrated in subsection 3.2, only language 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.

```
(class sample-calling-sequence object
  (method void register-pcd ()
    (meta register-one-pcd 'calling-sequence
                           calling-sequence-pcd?)
    (super register-pcd))
  (method boolean calling-sequence-pcd?  ...)
```

Application programmers may override the `register-pcd` method defined in the `object` class. To call procedures defined in the framework provided by X-ASB, programmers may use the `meta` call. 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 metaobject protocols in reflective OOP languages. Using reflective mechanisms, application programmers can extend JPMs in order to deal with unanticipated application-specific requirements in the process of software evolution.

5. EFFECTIVENESS IN SOFTWARE EVOLUTION

As mentioned in previous sections, X-ASB is effective in unanticipated software evolution. We may face new kinds of crosscutting concerns, which cannot be handled by existing JPMs, in the process of software evolution. New kinds of JPMs will be needed when we face the following situations: 1) we want to extend existing JPMs slightly in order to adapt to application-specific purposes; 2) we want to use more than one JPM simultaneously. As an example of the first case, we showed a JPM development process of the extended PA weaver in section 3. The extended PA weaver was developed modularly to support context-sensitive calling sequences that are not provided by the original PA weaver. As an example of the second case, it may be necessary to combine PA-like JPMs with TRAV-like JPMs. This can be realized using the level 2 framework of X-ASB. Using X-ASB, we can extend new kinds of JPMs according to software evolution.

6. RELATED WORK

Shonle, Lieberherr, and Shah propose an extensible domain-specific AOP language XAspect that adopts plug-in mechanisms[8]. Adding a new plug-in module, we can use a new kind of aspect-oriented facility. CME (Concern Manipulation Environment)[3], the successor of Hyper/J, adopts an approach similar to XAspect.

Although pointcut languages play important roles in AOP paradigms, current AOP languages do not provide sufficient kinds of pointcut constructs. In an effort to address this problem, several previous investigations have attempted to enrich the pointcut constructs. Kiczales emphasizes the necessity of new kinds of pointcut constructs such as `pcflow` (predictive control flow) and `dflow` (data flow)[5]. Gybels and Brichau point out problems of current pointcut languages from the viewpoint of the software evolution, and propose robust pattern-based pointcut constructs using logic programming facilities[4]. These approaches introduce new pointcut constructs in order to deal with new kinds of crosscutting concerns. However, adopting these approaches, we need to add another pointcut construct 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 `Josh`[2] in which programmers can define a new pointcut construct as a boolean function. Using X-ASB, we can add not only new pointcut constructs but also new kinds of join points and advice.

7. CONCLUSION

The paper proposed mechanisms for extending JPMs in order to support unanticipated software evolution. Using X-ASB, we can introduce new kinds of JPMs when we face new kinds of crosscutting concerns that cannot be handled by existing JPMs.

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, Science, Sports and Culture.

9. REFERENCES

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

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

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

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

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

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

[7] 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.

[8] 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.