Metamodel Access Protocols for Extensible Aspect-Oriented Modeling

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

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

Shinji Sano, Yusaku Maeno, Satoshi Murakami
Kyushu Institute of Technology
Fukuoka, Japan
{sano,maeno,msatoshi}@minnie.ai.kyutech.ac.jp

Abstract

Aspect orientation is important not only at the programming-level but also at the modeling-level. We previously proposed an aspect-oriented modeling language called AspectM for managing modeling-level aspects. Although AspectM provides basic modeling facilities for a modeler, the language constructs cannot be extended. In this paper, we propose a mechanism called metamodel access protocol (MMAP) that allows an application modeler to access and modify the AspectM metamodel. MMAP consists of metamodel extension points, extension operations, and primitive predicates for defining pointcut designators. MMAP enables a modeler to represent application-specific crosscutting concerns.

1. Introduction

Aspect-oriented programming (AOP)[9][4] can separate crosscutting concerns from primary concerns. In major AOP languages such as AspectJ[10], crosscutting concerns including logging, error handling, and transaction are modularized as aspects and they are woven to primary concerns. AOP is based on join point mechanisms (JPM)[11] consisting of join points, a means of identifying join points (pointcut), and a means of semantic effect at join points (advice). In AspectJ, program points such as method execution are detected as join points, and a pointcut designator extracts a set of join points related to a specific crosscutting concern from all join points. A compiler called a weaver inserts advice code at the join points selected by pointcut designators.

Recently, aspect orientation has been proposed for coping with concerns at the early stages of the software development phases including requirements analysis, domain analysis, and architecture design phases. Aspects at the modeling-level are mainly applied to not behavior as in AOP but a static model structure such as UML diagrams.

We previously proposed a UML-based aspect-oriented modeling language called AspectM[19] for managing modeling-level aspects. Using AspectM, a modeler can represent crosscutting concerns without considering details of implementation languages and platform because a model can be translated into source code by the AspectM model compiler. Although AspectM provides major JPMs, there might be situations in which a modeler wants to define a JPM specific to an application. For example, a modeler wants to capture a group of methods that are targets of an application-specific logging or transaction. While some kinds of application-specific concerns can be represented by using UML profiling mechanisms, there are applications that need metamodel extension. However, current aspect-oriented modeling languages including AspectM do not allow a modeler to extend JPMs.

In this paper, we introduce a mechanisms called metamodel access protocol (MMAP) that allows an application modeler to access and modify the AspectM metamodel, an extension of the UML metamodel. The mechanism enables a modeler to define a new JPM that includes application-specific join points, pointcut designators, and advice.

The remainder of this paper is structured as follows. In Section 2, we demonstrate the necessity of application-specific extension. In Section 3, we introduce the concept of MMAP. In Section 4, we illustrate an implementation method. In Section 5, we introduce related work. Lastly, in Section 6, we conclude the paper.
2. Motivation

We briefly excerpt an aspect-oriented modeling language AspectM from our previous work[19]. We also demonstrate the necessity of application-specific aspect orientation.

2.1. AspectM

2.1.1 Aspect orientation at the modeling-level

Although JPMs have been proposed as a mechanism at the programming-level, they can be applied to the modeling-level. Figure 1 shows an example of the modeling-level aspect orientation. In Figure 1, a class is regarded as a join point. The pointcut definition ‘classA || classB’ extracts the two classes classA and classB from the three join points classA, classB, and classC. Add new attributes and add new operations are regarded as advice. In Figure 1, new attributes and operations are added to the two classes classA and classB. As shown here, aspects at the modeling-level are applied to not behavior as in AOP but a static model structure.

AspectM supports six kinds of modeling-level JPMs: PA (pointcut & advice as in AspectJ[10]), CM (composition as in HyperJ[17]), NE (new element), OC (open class as in AspectJ inter-type declaration), RN (rename), and RL (relation). Figure 1 is an example of OC. Table 1 shows the outline of these JPMs.

2.1.2 Language features

AspectM provides the two kinds of aspects: an ordinary aspect and a component aspect. A component aspect is a special aspect used for composing aspects. An aspect can have parameters for supporting generic facilities.

Figure 2 shows the AspectM diagram notations and the corresponding XML formats. The notations of aspect diagrams are similar to those of UML class diagrams. A diagram is separated into three compartments: 1) aspect name and JPM type, 2) pointcut definitions, and 3) advice definitions. An aspect name and a JPM type are described in the first compartment. A JPM type is specified using a stereotype. Pointcut definitions are described in the second compartment. Each of them consists of a pointcut name, a join point type, and a pointcut body. In pointcut definitions, we can use three designators: cname (class name matching), aname (attribute name matching), and oname (operation name matching). We can also use three logical operations: && (and), || (or), and ! (not). Advice definitions are described in the third compartment. Each of them consists of

<table>
<thead>
<tr>
<th>JPM type</th>
<th>Join point type</th>
<th>Advice type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA (pointcut &amp; advice)</td>
<td>operation</td>
<td>before, after, around</td>
</tr>
<tr>
<td>CM (composition)</td>
<td>class</td>
<td>merge-by-name</td>
</tr>
<tr>
<td>NE (new element)</td>
<td>class diagram</td>
<td>add-class, delete-class</td>
</tr>
<tr>
<td>OC (open class)</td>
<td>class</td>
<td>add-operation, delete-operation</td>
</tr>
<tr>
<td>RN (rename)</td>
<td>class, operation, attribute</td>
<td>rename</td>
</tr>
<tr>
<td>RL (relation)</td>
<td>class</td>
<td>add-inheritance, delete-inheritance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add-aggregation, delete-aggregation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add-relationship, delete-relationship</td>
</tr>
</tbody>
</table>

Table 1. JPMs in AspectM

Figure 1. Modeling-level aspect orientation

Figure 2. AspectM notation
an advice name, a pointcut name, an advice type, and an advice body. A pointcut name is a pointer to a pointcut definition in the second compartment. An advice is applied at join points selected by a pointcut.

2.2. Problems to be tackled

Although AspectM provides basic JPMs for aspect-oriented modeling, a modeler cannot define application-specific JPMs. It would be better for a modeler to describe a model as shown in Figure 3 (the class diagram is cited from [6]). Application-specific model elements are denoted by stereotypes that are not merely annotations but elements introduced by metamodel extension.

The model in Figure 3 describes an invoice processing system comprised of two kinds of application-specific distributed components: DCEntityContract for defining the contract of a distributed entity component and DCControllerContract for defining the contract of a distributed controller component. The model also includes an application-specific JPM DCLogger that adds log operations to DCEntityContracts whose UniqueId is not assigned by users. DCLogger consists of application-specific pointcut designators and advice. DCEntityContract can be regarded as an application-specific join point.

Figure 3. An example of application-specific model

It is not necessarily easy to describe aspects such as DCLogger by using stereotypes as merely annotations because associations among stereotypes cannot be specified. Without extending a metamodel, we cannot specify the following fact: DCEntityContract must have a UniqueId whose tag is isUserAssigned. If we define an aspect based on fragile stereotypes that lack consistency, the aspect might introduce unexpected faults because the aspect affects many model elements. It is necessary to adopt metamodel extension for describing application-specific models precisely.

In this paper, we propose a method for constructing this kind of application-specific models by introducing the concept of MMAP. There are many situations that need application-specific JPMs—for example, application-specific logging as in Figure 3, application-specific resource management, application-specific transaction, and so on.
3. Metamodel access protocol

3.1. Background

There are two approaches for extending model elements in UML. The first is a lightweight approach using UML profiles. The second is a heavyweight approach that extends the UML metamodel by using MOF (Meta Object Facility), a language for modeling a metamodel.

UML profiling mechanisms use stereotypes for introducing rich vocabularies such as UML Profiles for Enterprise Distributed Object Computing (EDOC). While an application modeler can easily introduce some kinds of application-specific JPMs—for example, pointcut definitions using stereotypes such as UniqueId—there are situations in which stereotypes as a mere annotation mechanism are insufficient: the typing of tags is weak; and we cannot declare new associations among UML metamodel elements[6]. For example, as explained in 2.2, stereotypes as annotations cannot describe relations among application-specific model elements including DCEntityContract, DCControllerContract, UniqueId, and DCLogger: the relations among them are described in the metamodel as shown in Figure 4.

On the other hand, the MOF approach is very strong because all of the metamodel elements can be extended. However, it is not easy for an application modeler to extend the UML metamodel by using the full power of the MOF.
MMAP proposed in this paper is a middle weight approach that restricts available extension by MOF. MMAP aims at introducing an application-specific JPM that cannot be described by UML profiling mechanisms. An application modeler can access and modify the AspectM metamodel by using protocols exposed by MMAP. The AspectM metamodel is an extension of the UML metamodel. The target of MMAP is not a tool developer that needs full access to the AspectM metamodel but an application modeler that wants to introduce rich vocabulary at small cost. Although model elements introduced by MMAP are denoted by stereotypes for convenience, these stereotypes are not merely annotations.

The idea of MMAP originates in the mechanisms of extensible programming languages, such as metaobject protocol (MOP) [8] and computational reflection [15] [12]. We think that the idea of MOP is useful at the modeling level although the targets of MOP are different from those of MMAP: the former focuses on execution behavior, and the latter focuses on model structures.

### 3.2. Metamodel

The AspectM metamodel is defined by modifying the UML metamodel. OMG defines the four-level metamodel hierarchy consisting of M0, M1, M2, and M3: M0 contains the data such as object instances; M1 contains application models described by a modeling language such as UML; M2 contains metamodels that define modeling languages; M3 contains metatamodels that define metamodels in M2. For example, a class diagram and the UML metamodel reside at the level M1 and M2, respectively. MOF resides at the level M3.

Figure 4 shows the part of the AspectM metamodel and an example of extension with MMAP. This metamodel resides at the level M2. By introducing this metamodel, we can define an aspect in a UML diagram that resides at the M1 level.

### 3.3. Protocols

MMAP is exposed for a modeler who creates a model at the M1 level to access the AspectM metamodel at the M2 level. MMAP is comprised of extension points, extension operations, and primitive predicates for navigating the AspectM metamodel. An extension point is an AspectM metamodel element that can be extended by inheritance. An extension operation is a modeling activity allowed at the exposed extension points.

Table 2 and 3 show protocols exposed by MMAP. The former is a list of extension points and operations. The latter is a list of primitive predicates that can navigate the AspectM metamodel and define pointcut designators.

<table>
<thead>
<tr>
<th>Extension point</th>
<th>Extension operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>define subclasses,</td>
</tr>
<tr>
<td></td>
<td>add attributes to subclasses,</td>
</tr>
<tr>
<td></td>
<td>create associations among subclasses</td>
</tr>
<tr>
<td>Attribute</td>
<td>the same as above</td>
</tr>
<tr>
<td>Operations</td>
<td>the same as above</td>
</tr>
<tr>
<td>PointcutAndAdvice (PA)</td>
<td>the same as above</td>
</tr>
<tr>
<td>Composition (CM)</td>
<td>the same as above</td>
</tr>
<tr>
<td>NewElement (NE)</td>
<td>the same as above</td>
</tr>
<tr>
<td>OpenClass (OC)</td>
<td>the same as above</td>
</tr>
<tr>
<td>Rename (RN)</td>
<td>the same as above</td>
</tr>
<tr>
<td>Relation (RL)</td>
<td>the same as above</td>
</tr>
</tbody>
</table>

Table 2. Extension points and operations

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta-class-of(mc, c)</td>
<td>mc is a metaclass of c</td>
</tr>
<tr>
<td>member-of(m, c)</td>
<td>m is a member of a class c</td>
</tr>
<tr>
<td>value-of(v, a)</td>
<td>v is value of an attribute a</td>
</tr>
<tr>
<td>super-class-of(c1, c2)</td>
<td>c1 is a superclass of c2</td>
</tr>
<tr>
<td>related-to(c1, c2)</td>
<td>c1 is related to c2</td>
</tr>
</tbody>
</table>

Table 3. Primitive predicates for defining pointcut designators

The design of MMAP is similar to that of application frameworks in which hot-spots should be exposed. By using MMAP, an application modeler need not to redefine the AspectM metamodel. The modeler has only to extend these hot-spots by using protocols.

### 3.3.1 Extension points and operations

We can extend the AspectM metamodel as shown in Figure 4 by applying extension operations such as define subclasses / add attributes to the subclasses at the exposed extension points including Class, Attribute, Operation and OpenClass. By adopting this extended metamodel, we can describe an application-specific UML model as shown in Figure 3.

This metamodel introduces the following application-specific model elements: application-specific join points including DCEntityContract and DCControllerContract, an attribute UniqueId, and operations such as DCLoggerOperation; an application-specific advice DCLogger that can add an application-specific operation DCLoggerOperation to application-specific join points.

In our MMAP, Advice/AdviceBody are not exposed as extension points because this extension need new weaver modules that can handle new kinds of advice. It is hard for
a modeler to develop a new weaver module. On the other hand, it is relatively easy to extend or synthesize existing advice mechanisms including PA, CM, NE, OC, RN, and RL. For these reasons, our MMAP does not allow a modeler to extend Advice/AdviceBody.

Although other metamodel elements such as association might be candidates of extension points, they are not included in our MMAP because their usefulness for application-specific extension is not yet clear.

### 3.3.2 Predicates for pointcut designators

A modeler does not have to extend Pointcut or PointcutBody because pointcut designators can be defined by composing primitive predicates shown in Table 3. The following is a definition of application-specific pointcut designator appearing in Figure 3:

```
define pointcut
DCEntityContract_UniqueId_isUserAssigned(c):
  meta-class-of("DCEntityContract", c) &&
  member-of(a, c) &&
  meta-class-of("UniqueId", a) &&
  member-of("isUserAssigned", "UniqueId") &&
  value-of("true", "isUserAssigned")
```

This pointcut designator selects all classes that match the following conditions: 1) the metaclass is DCEntityContract; 2) the value of the isUserAssgned is true. In case of Figure 3, the negation of this pointcut designator selects the two classes Customer and Invoice. It is necessary to distinguish quantification of pointcut designators from that of predicates. The latter is existential. A defined pointcut designator represents all elements that satisfy the right-hand side predicates.

By using the primitive predicates, we can define built-in pointcut designators as follows:

```
define pointcut
cname(c):
  meta-class-of("Class", c) &&
  member-of("Name", "Class") &&
  value-of(c, "Name")
```

### 4. Implementation

We have developed a prototype tool for supporting AspectM. The tool consists of a model editor and a model compiler. The model editor provides facilities for editing UML and aspect diagrams. The model editor is developed using Eclipse Modeling Framework (EMF)[5], a tool that generates a model editor from a metamodel. The model editor can save diagrams in the XML format. The model compiler is implemented as an XML transformation tool because UML class diagrams can be represented in the XML format.

Our prototype tool includes the core part of MMAP. The AspectM metamodel and application models are transformed to a set of Prolog predicates. We can access and modify these model elements by using predicates exposed by MMAP.

We also plan to develop a reflective model editor that allows a modeler to not only edit application models but also extend, modify, and customize the AspectM metamodel. Figure 5 illustrates the concept of a reflective model editor. This idea is similar to an edit-time metaobject protocol (ETMOP)[3] proposed by A.D.Eisenberg and G.Kiczales. ETMOP runs as part of a code editor and enables metadata annotations to customize the rendering and editing of code. Using the AspectM reflective model editor, a modeler can edit a model such as Figure 3 by extending the AspectM metamodel as shown in Figure 4.

### 5. Related work

Extensible AOP languages can consolidate the advantages of domain-specific and general-purpose languages because programmers can customize language features. H.Masuhara and G.Kiczales defined a three-part modeling framework that explains a common structure in different major JPMs such as PA, CM, and OC[13]. Based on the three-part modeling framework, N.Ubayashi et al. proposed a parameterized interpreter that takes several parameters to cover different JPMs[18]. The interpreter is helpful in rapid-prototyping a new AOP mechanism or a reflective AOP system that supports different mechanisms. M.Shonle, K.Lieberherr, and A.Shah proposed an extensible domain-specific AOP language called XAspect that adopts plug-in mechanisms[14]. E.Tanter and J.Noye proposed a versatile kernel for multi-language AOP[16]. S.Chiba et al. proposed Josh[1] that allows programmers to define a new pointcut designator as a boolean function. AspectM covers major
JPMs including PA, CM, and OC. Moreover, application-specific AOP features can be introduced by using MMAP proposed in this paper.

Domain-specific aspect orientation is important not only at the programming stage but also at the modeling stage. An approach for supporting domain-specific aspect-oriented modeling is proposed by J.Gray[7]. Logic programming facilities and queries using these facilities will be useful for defining domain-specific pointcuts[20]. M.Eichberg, M.Mezini, and K.Ostermann investigated the use of XQuery for specification of pointcuts[2]. Since a domain can be considered as a set of application families, MMAP is useful for domain-specific aspect orientation by defining JPMs common to related applications.

6. Conclusion

In this paper, we proposed MMAP that enables a modeler to define a new JPM that represents application-specific crosscutting concerns. Although the current MMAP might need refining, we believe that our approach is the first step towards extensible aspect-oriented modeling based on metamodel access protocols.

References


