Model Compiler Construction Based on Aspect-Oriented Mechanisms

Naoyasu Ubayashi[1], Tetsuo Tamai[2],
Shinji Sano[1], Yusaku Maeno[1], and Satoshi Murakami[1]

[1] Kyushu Institute of Technology, Japan
[2] University of Tokyo, Japan
{ubayashi,tamai}@acm.org
{sano,maeno,msatoshi}@minnie.ai.kyutech.ac.jp

Abstract. Model-driven architecture (MDA) aims at automating software design processes. Design models are divided into platform-independent models (PIMs) and platform-specific models (PSMs). A model compiler transforms the former models into the latter automatically. We can regard PIMs as a new kind of reusable software component because they can be reused even if a platform is changed. However, a generated PSM is useless if it does not satisfy system limitations such as memory usage and real-time constraints. It is necessary to allow a modeler to customize transformation rules because model modifications for dealing with these limitations may be specific to an application. However, current model compilers do not provide the modeler sufficient customization methods. In order to tackle this problem, we propose a method for constructing an extensible model compiler based on aspect orientation, a mechanism that modularizes crosscutting concerns. Aspect orientation is useful for platform descriptions because it crosscuts many model elements. A modeler can extend model transformation rules by defining new aspects in the process of modeling. In this paper, an aspect-oriented modeling language called AspectM (Aspect for Modeling) for supporting modeling-level aspects is introduced. Using AspectM, a modeler can describe not only crosscutting concerns related to platforms but also other kinds of crosscutting concerns. We believe that MDA is one of the applications of aspect-oriented mechanisms. The contribution of this paper is to show that a model compiler can actually be constructed based on aspect-oriented mechanisms.

1 Introduction

Model-driven architecture (MDA) aims at automating software design processes. Design models described in Unified Modeling Language (UML) are divided into platform-independent models (PIMs) and platform-specific models (PSMs). A model compiler transforms the former models into the latter automatically. The current MDA primarily focuses on platform-related issues. However, model transformations are not limited to these concerns, as in the case of application-specific optimization. A PSM generated by a model compiler is useless if the PSM does

not satisfy system limitations such as memory usage and real-time constraints. It is necessary to allow a modeler to customize transformation rules because model modifications for dealing with these limitations may be specific to an application. It would be useful to apply the idea of active libraries[5] to model compiler construction. However, most current model compilers support only specific kinds of platforms, and do not provide the modeler sufficient customization methods.

This paper proposes a method for constructing an extensible model compiler based on aspect orientation[12] in order to tackle the above problem. Aspect orientation is a mechanism that modularizes crosscutting concerns as aspects. Platform descriptions also can be regarded as crosscutting concerns. For example, descriptions for conforming a model to a specific database middleware cut across many elements in a model. There are several reasons why adopting aspect-oriented mechanisms for describing database concerns is useful: persistence can be modularized; persistence aspects can be reused; and applications can be developed unaware of the persistent nature of the data[18]. The approach of [18] is effective not only at the programming level but also at the modeling level. In this paper, an aspect-oriented modeling language called AspectM (Aspect for modeling) is introduced for supporting modeling-level aspects. Using AspectM, a modeler can describe not only crosscutting concerns related to platforms but also other kinds of concerns related to model transformation. That is, a modeler can extend model transformation rules by defining new aspects in the process of modeling: that is, defining model transformation rules at the same level of ordinary modeling. Using AspectM, we can realize not only MDA but also techniques for supporting early aspects and crosscutting properties at the requirement-related and architectural levels[6]. MDA and aspect orientation are not different software development principles; rather, we believe that MDA is an application of aspect-oriented mechanisms. The contribution of this paper is to show that a model compiler can be actually constructed based on aspect-oriented mechanisms.

The remainder of this paper is structured as follows. In Section 2, we illustrate the process of model transformation using a simple example. In Section 3, we propose a method for model transformations based on aspect-oriented mechanisms. We introduce AspectM for supporting the method, and provide a technique for implementing AspectM in Section 4. We show a model transformation example using AspectM in Section 5. In Section 6, we evaluate AspectM qualitatively based on our experience. In Section 7, we introduce some related work, and discuss future directions of this research. Section 8 concludes the paper.

2 Motivation

Here, we illustrate typical model transformation steps in MDA, show how platform-specific concerns cut across model elements, and demonstrate how aspect-oriented mechanisms can be applied to describe these crosscutting concerns.

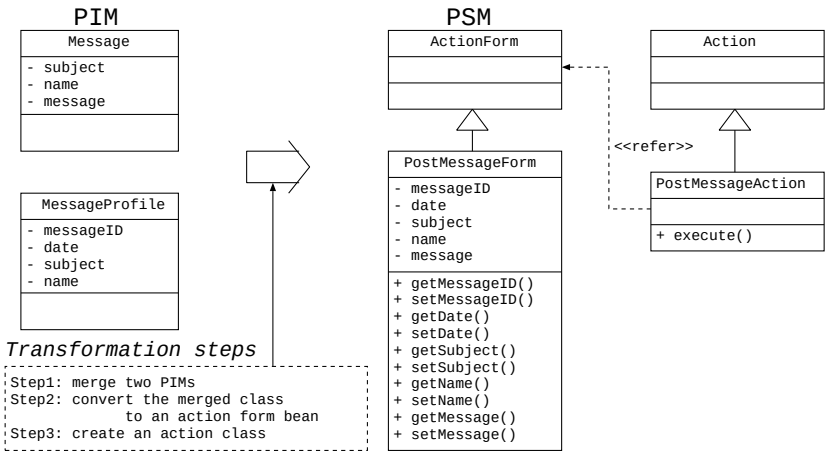


Fig. 1. An example of a model transformation

2.1 Model transformation steps in MDA

The steps of model transformation can be explained using the following simple bulletin board system as an example: *a user submits a message to a bulletin board, and the system administrator observes administrative information such as daily message traffic.* This system must be developed using the web application framework called Struts[22].

We define PIMs that do not depend on a specific platform, and transform these PIMs into PSMs targeted to Struts, the platform of this system. Figure 1 illustrates this transformation process. There are two PIMs in this example[3]. One is the `Message` class, and the other is the `MessageProfile` class. The former is a PIM defined from the viewpoint of a user. The latter, which includes administrative information such as message id and date, is a PIM defined from the viewpoint of a system administrator. Although these PIM classes represent different viewpoints in the system, the substance of the classes should be the same. All attributes included in the `Message` class and the `MessageProfile` class are necessary for handling a message. The following shows the steps of transformation of PIMs to a PSM.

Step 1: The two PIM classes, `Message` and `MessageProfile`, are merged into a single class whose name is `PostMessage`. Attributes/Operations that have the same name (signature) are merged into a single attribute/operation.

Step 2: In Struts, a request from a web browser is stored in an action form bean class. The `PostMessage` class is transformed to an action form bean class. First, the name of the `PostMessage` class is changed to `PostMessageForm` because an action form bean class must have a name ending with the string

[3] In general, PIMs and PSMs are described as sets of UML diagrams. In this example, we use only class diagrams for simplicity.

Form. Next, the parent class of the `PostMessageForm` is set to the `ActionForm` framework class because it is specified in Struts that a bean class must inherit the `ActionForm` class. After that, a set of accessors (setter/getter) is added to the `PostMessageForm` class. The transformations in Step 2 are needed for every data request. That is, the transformations cut across classes related to the requested data.

Step 3: In Struts, an action logic that handles a request from a web browser is defined as the `execute` operation in an action class. First, the action class `PostMessageAction` is created, and its parent class is set to the `Action` class prepared in Struts. Next, the `execute` operation is added to the `PostMessage-Action` class. The `execute` operation gets the data of the request from the corresponding action form bean class, and executes a business logic.

We can implement a model compiler that supports the above transformation steps because each step is clearly defined. We can also develop a new model compiler that supports other platforms. A series of PSMs can be generated from a single PIM by applying different model compilers. MDA enables us to shift from code-centric product-line engineering (PLE)[5] to model-centric PLE.

2.2 Advantages of introducing aspect orientation

In this paper, we introduce aspect-oriented mechanisms for describing model transformation rules. As pointed out in the above, platform-specific descriptions are one of the crosscutting concerns that can be well dealt with by aspect orientation. Although MDA and aspect orientation at the modeling level are considered different technologies, they are closely related, as we claim in this paper. Applying aspect orientation to model compiler construction, we obtain the following advantages: a modeler can extend model transformation rules by defining new aspects in the process of modeling; a modeler can describe not only crosscutting concerns related to platforms but also other kinds of crosscutting concerns including optimization, persistence, and security; and aspect orientation can be applied at the modeling level, at which design information can be used to represent crosscutting concerns.

3 Applying aspect orientation to model compiler construction

3.1 Aspect orientation at the modeling level

Aspect-oriented programming (AOP), a modularization mechanism for separating crosscutting concerns, is based on the join point model (JPM) consisting of join points, pointcuts, and advice. Program execution points including method invocations and field access points are detected as join points, and a pointcut 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 definitions.

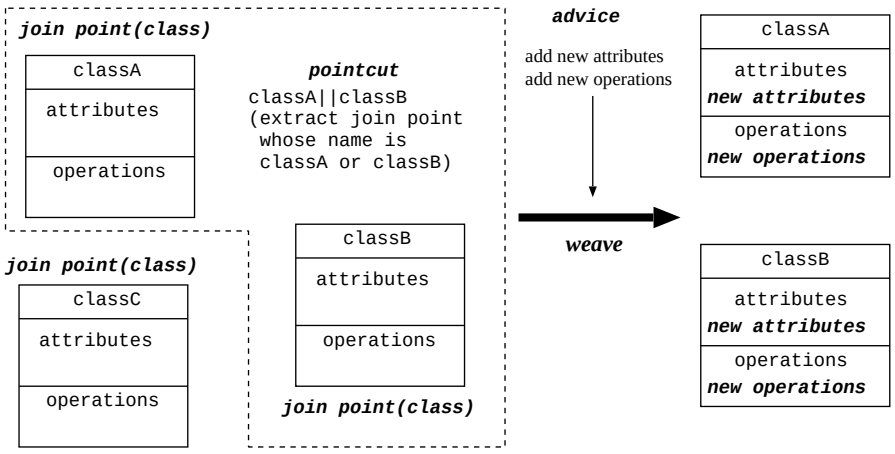


Fig. 2. Aspect orientation at the modeling level (example)

Although JPMs have been proposed as a mechanism at the programming level, they can be applied to the modeling level as shown in Figure 2. In this example, 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 `class A`, `classB`, and `classC`. Model transformations such as *add new attributes* and *add new operations* are regarded as advice. In Figure 2, new attributes and operations are added to the two classes, `classA` and `classB`.

3.2 JPMs for model transformations

There has been research supporting modeling-level aspect orientation based on a specific AOP language such as AspectJ[13]: an aspect at the modeling level is converted to an aspect in AspectJ[21]. However, there are problems with these approaches: a PSM is limited to a specific AOP language; most current AOP languages are based on a few fixed set of JPMs; and we cannot separate cross-cutting concerns that cannot be separated by current AOP languages. Indeed, there are several kinds of JPMs as shown in [14]. In order to deal with this problem, multiple JPMs should be supported at the modeling level, and these JPMs should not correspond to specific kinds of AOP languages.

AspectM, an aspect-oriented modeling language proposed in this paper, supports six kinds of JPMs: PA (pointcut & advice), CM (composition), NE (new element), OC (open class), RN (rename), and RL (relation). Table 1 shows model transformation types and corresponding JPMs. With aspect composition based on these JPMs, the model transformation in Section 2 can be realized as shown in Table 2. Model elements including classes, methods, and relations specific to Struts are woven into the original PIMs.

PA is an AspectJ-like JPM. A join point is a method execution, and advice changes a behavior at join points selected by a pointcut. Three kinds of advice

No	Model transformation type	PA	CM	NE	OC	RN	RL
1	change a method body	◯					
2	merge classes		◯				
3	add/delete classes			◯			
4	add/delete operations				◯		
5	add/delete attributes				◯		
6	rename classes					◯	
7	rename operations					◯	
8	rename attributes					◯	
9	add/delete inheritances						◯
10	add/delete aggregations						◯
11	add/delete relationships						◯

Table 1. JPMs for model transformation

Step	Model transformation	PA	CM	NE	OC	RN	RL
step 1	1-1) merge `Message` and `MessageProfile` into `PostMessage`		◯				
step 2	2-1) rename `PostMessage` to `PostMessageForm`					◯	
	2-2) add an inheritance relation between `ActionForm` and `PostMessageForm`						◯
	2-3) add accessors to `PostMessageForm`				◯		
step 3	3-1) create an action class `PostMessageAction`			◯			
	3-2) add an inheritance relation between `Action` and `PostMessageAction`						◯
	3-3) add the `execute` method to `PostMessageAction`				◯		
	3-4) add the body of the `execute` method	◯					

Table 2. Model transformation steps for Struts

can be described: `before` (a pre-process is added), `after` (a post-process is added), and `around` (a process is replaced). PA is used when we want to add platform-specific logics to PIMs. CM is a Hyper/J-like JPM[4]. In this case, a join point is a class, and advice merges classes selected by a pointcut: operations with the same name are merged into a single operation, and attributes with the same name are merged into a single attribute. CM is used in the case of converting multiple PIM classes to a single PSM class. NE is a JPM for adding a new model element to a UML diagram. In this case, a join point is a UML diagram such as a class diagram. Advice adds a new class to a class diagram selected by a pointcut. NE can be used to add a platform specific class to PIMs. OC is a JPM for realizing the facility of an open class. In this case, a join point is a class, and advice inserts operations or attributes. OC, which is similar to an inter-type declaration in AspectJ, is used in the case of adding platform-specific operations or attributes to PIMs. Figure 2 in Section 3.1 is an example of an OC. RN is a JPM for changing a name, in which a join point is a class, an operation, and an attribute. Advice changes the names of classes, operations, and attributes selected by a pointcut. RN is used for following the naming conventions specified in a platform. RL is a JPM for changing the relation between two classes, in which

JPM type	Join point type	Advice type
PA	operation	before, after, around
CM	class	merge-by-name
NE	class diagram	add-class, delete-class
OC	class	add-operation, delete-operation add-attribute, delete-attribute
RN	class, operation, attribute	rename
RL	class	add-inheritance, delete-inheritance add-aggregation, delete-aggregation add-relationship, delete-relationship

Table 3. Types of JPM, join point, and advice

case, a join point is a class, and advice adds an inheritance, an aggregation, and a relationship between two classes selected by a pointcut. There is a case that a class must inherit a specific class defined in an application framework such as Struts. RL is used in this situation.

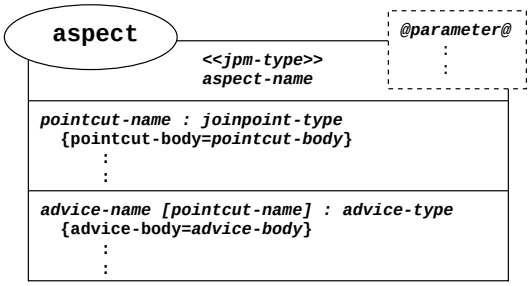
4 AspectM

AspectM is an aspect-oriented modeling language that supports the six JPMs introduced in Section 3. In AspectM, an aspect can be described in either a diagram or an XML (eXtensible Markup Language) format. AspectM is defined as an extension of the UML metamodel. Figure 3 shows the AspectM diagram notations and the corresponding XML formats. AspectM is not only a diagram language but also an XML-based AOP language. In this section, we show the syntax of AspectM, which has two aspects: an ordinary aspect and a component aspect. A component aspect is a special aspect used for composing aspects. In this paper, we use simply the term *aspect* when we need not to distinguish between an ordinary aspect and a component aspect. An aspect can have parameters for supporting generic facilities. By filling parameters, an aspect for a specific purpose is generated. Using these kinds of aspects, a set of transformation steps can be described as a generic software component.

4.1 Notation

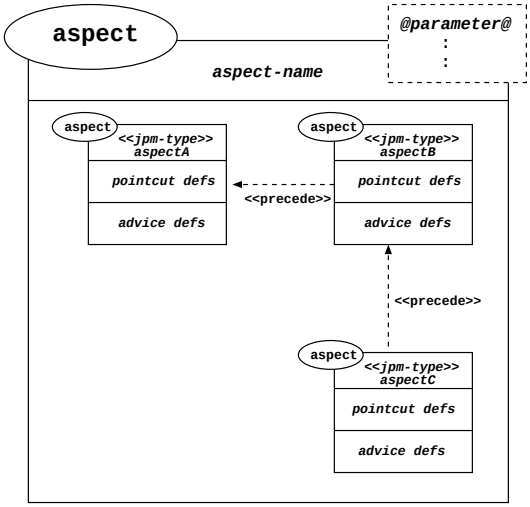
The notations of aspect diagrams are similar to those of UML class diagrams. An oval at the upper left portion of a diagram indicates that the diagram represents an aspect. A box at the upper right indicates parameters. This box can be omitted when there is no parameter. An aspect with parameters is called a *template*. Italic text in a diagram must be specified by a modeler. Types of JPMs (*jpm-type*), join points (*joinpoint-type*), and advice (*advice-type)* are shown in Table 3.

Diagrams of ordinary aspects are separated into three compartments: 1) aspect name and JPM type, 2) pointcut definitions, and 3) advice definitions. An



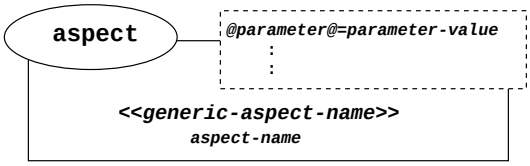

```
<aspect name=aspect-name type="ordinary"
        jpm=jpm-type>
  [<params>
    { <param> @parameter@ </param> } *
   </params>]
  { <pointcut name=pointcut-name
              type=joinpoint-type>
      <pointcut-body>
        pointcut-body
      </pointcut-body>
    </pointcut> } +
  { <advice name=advice-name
            type=advice-type
            ref-pointcut=pointcut-name>
     <advice-body>
        advice-body
     </advice-body>
    </advice> } +
</aspect>
```

a) ordinary aspect




```
<aspect name=aspect-name type="component" >
  [<params>
    { <param> @parameter@ </param> } *
   </params>]
  { definition of ordinary aspect
       or
    definition of component aspect} +
  { <aspect-precedence>
     <from>aspect-name</from>
     <to>aspect-name</to>
    </aspect-precedence> } +
</aspect>
```

b) component aspect




```
<aspect name=aspect-name
        template="template-name" >
  {<set-param name="@parameter@">
     parameter-value
   </set-param> } +
</aspect>
```

c) parameter setting

Fig. 3. AspectM diagram notations and XML formats

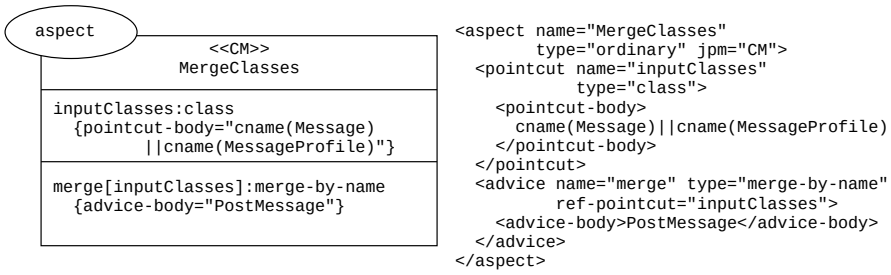


Fig. 4. Example of aspect notation

aspect name and a JPM type are described in the first compartment. A JPM type is specified using a stereo type. 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 predicates: `cname` (class name matching), `aname` (attribute name matching), and `oname` (operation name matching). We can also use three logical operations: && (and), || (or), and ! (not). The following are examples of pointcut definitions: '`oname(setX)` || `oname(setY)`' (two operations `setX` and `setY` are selected from all join points); '`!aname(attribute*)`' (attributes not starting with a string `attribute` are selected); '`cname(classA)` || `cname(classB)`' (two classes `classA` and `classB` are selected); and '`cname(class*)` &&`oname(set*)`' (operations, which belong to classes starting with a string `class` and start with a string `set`, are selected if joinpoint-type is operation). Although we support only predicates for name matching in the current AspectM, we plan to support predicates including `is-attribute-of(`*class*`)`, `is-operation-of(`*class*`)`, `is-superclass-of(`*class*`)`, and `is-subclass-of(`*class*`)`. Advice definitions are described in the third compartment. Each of them consists of 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. Advice is applied at join points selected by the pointcut. The left side of Figure 4 shows a transformation rule corresponding Step 1 in Section 2: the JPM type is CM; the two classes `Message` and `MessageProfile` are join points selected by the pointcut definition; and the merge-by-name type advice is applied at these join points.

Diagrams of component aspects are separated into two compartments: 1) aspect name, and 2) a set of ordinary aspects or component aspects. A component aspect consists of the aspects specified in the second compartment. A stereo type `<<precede>>` indicates the precedence of aspects as shown in Figure 3 b). This is important when multiple aspects are applied to the same join points.

By filling parameter values, an aspect for a specific purpose is generated. The name of a template is specified in a stereo type.

An aspect can be represented in XML format as shown in the right side of Figure 3. The notations [] and {} show an option and a repetition, respectively. The notations $*$ and $+$ in {} show an occurrence of more than zero and more

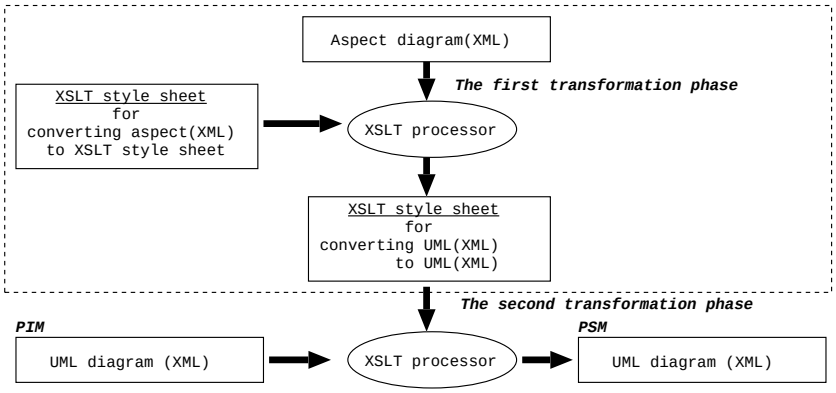


Fig. 5. Implementation of AspectM model compiler

than one. An aspect is represented by the `aspect` tag distinguished by the `type` attribute. A set of parameters is specified by the `params` tag. In an ordinary aspect, pointcuts and advice are specified by the `pointcut` tag and `advice` tag, respectively. In a component aspect, definitions of ordinary aspects or other component aspects are specified after parameter definitions. After that, precedences of aspects are specified using the `aspect-precedence` tag. Parameters are set using the `set-param` tag. The right side of Figure 4 shows the XML descriptions corresponding to the diagrams on the left.

4.2 Implementation

We have developed a prototype of AspectM. The tool for supporting AspectM consists of a model editor and a model compiler. The model editor facilitates editing UML and aspect diagrams. The model editor can save diagrams in the XML format. The model compiler can be implemented as an XML transformation tool because UML class diagrams can be represented in XML. The AspectM model compiler, which consists of two phases, transforms PIM classes into PSM classes as shown in Figure 5. The first transformation phase converts an aspect in the form of XML to an XSLT (XSL Transformation) style sheet with additional Java classes using an XSLT processor. The second transformation phase converts PIM classes in XML form to the corresponding PSM classes in XML form using the style sheet generated in the first transformation phase.

5 MDA with AspectM

5.1 Aspect descriptions for model transformations

Figure 4 shows Step 1 of transformation in the bulletin board system. In this step, the `MergeClasses` aspect, whose JPM type is CM, is defined for merging two PIM classes `Message` and `MessageProfile` into the `PostMessage` class.

Although the `MergeClasses` aspect in Figure 4 is useful, there is a problem in terms of reusability because the aspect cannot be applied to other models. The pointcut body and the advice body are specific to the bulletin board system. In order to deal with the problem, a generic mechanism can be used. The following is a generalized version of the `mergeClasses` in XML form. A string enclosed by '@' is a parameter.

```
;; generic MergeClasses aspect
<aspect" name="Generic-MergeClasses" type="ordinary" jpm="CM">
  <params>
    <param>@input-classes@</param>
    <param>@merged-class@</param>
  </params>
  <pointcut" name="inputClasses" type="class">
    <pointcut-body>@input-classes@</pointcut-body>
  </pointcut>
  <advice name="merge" adviceType="merge-by-name"
        ref-pointcut="inputClasses">
    <advice-body>@merged-class@</advice-body>
  </advice>
</aspect>
;; specific mergeClasses aspect
<aspect name="MergeClasses" template="Generic-MergeClasses">
  <set-param name="@input-classes@">
    cname(Message)||cname(MessageProfile)
  </set-param>
  <set-param name="@merged-class@">PostMessage</set-param>
</aspect>
```

Steps 2 and 3 can be also realized with the same approach. The following aspect describes a transformation step that adds an inheritance relation between the `ActionForm` class and an action form bean class. The `<relation>` is a tag for adding or deleting a relation such as an inheritance, an aggregation, or a relationship.

```
<aspect name="Generic-InheritActionForm" type="ordinary" jpm="RL">
  <params>
    <param>@sub-class@</param>
  </params>
  <pointcut name="super-sub-classes" type="class">
     cname(org.apache.struts.action.ActionForm)||cname(@sub-class@)
  </pointcut>
  <advice name="inherit-action-form" type="add-inheritance">
    <ref-pointcut>super-sub-classes</ref-pointcut>
    <advice-body>
      <relation>
        <end1>org.apache.struts.action.ActionForm</end1>
        <end2>@sub-class@</end2>
      </relation>
    </advice-body>
  </advice>
</aspect>
```

We can compose a set of related aspects within a component aspect. The following aspect, which generates an action form bean from PIM classes, composes aspects that describe Step 1 and 2.

```
<aspect name="Generic-Classes2ActionFormBean" type="component">
  <params>
    <param>@input-classes@</param>
    <param>@merged-class@</param>
  </params>
  <aspect name="MergeClasses" template="Generic-MergeClasses">
    <set-param name="@input-classes@">@input-classes@</set-param>
    <set-param name="@merged-class@">@merged-class@</set-param>
  </aspect>
  <aspect name="SetActionFormBeanName" template="Generic-SetActionFormBeanName">
    <set-param name="@class@">@merged-class@</set-param>
  </aspect>
  <aspect name="InheritActionForm" template="Generic-InheritActionForm">
    <set-param name="@sub-class@">concat(@merged-class@,"Form")</set-param>
  </aspect>
  <aspect name="AddAccessors" template="Generic-AddAccessors">
    <set-param name="@class@">concat(@merged-class@,"Form")</set-param>
  </aspect>
</aspect>
```

Four generic aspects are used for defining this component aspect. The definitions of the two generic aspects `Generic-SetActionFormBeanName` and `Generic-AddAccessors` are omitted here due to limitations of space. Sub-aspects are applied in the order of appearance when the `aspect-precedence` tags are omitted. The `concat` is a library function for concatenating two strings.

5.2 Extension of model transformations

Adopting AspectM, we can extend the functionality of the model compiler by adding aspect definitions. This extensibility is effective for defining application-specific model transformations. The following is the aspect that deletes the `date` attribute when the two classes `Message` and `MessageProfile` are merged.

```
<aspect name="DeleteAttribute" type="ordinary" jpm="OC">
  <pointcut name="postMessageClass" type="class">
    <pointcut-body>cname(PostMessage)</pointcut-body>
  </pointcut>
  <advice name="deleteDate" adviceType="delete-attribute"
         ref-pointcut="postMessageClass">
    <advice-body>date</advice-body>
  </advice>
</aspect>
```

This kind of aspect is useful for product-line engineering in which a variety of PSMs are generated from a single set of PIMs. A specific PSM, a model of a specific product, may have to be optimized in terms of memory resources. The above aspect, which eliminates the `date` attribute unused in a specific product, is applied after the `MergeClasses` aspect is applied. Using AspectM, a process of tuning up can be componentized as an aspect.

5.3 Descriptions for other crosscutting concerns

AspectM can also describe the type of crosscutting concern that AspectJ supports. The following is an aspect for logging setter method calls. `Log.write()` is a log writer.

```
<aspect name="LoggingSetter" type="ordinary" jpm="PA">
  <pointcut name="allSetter" type="method">
    <pointcut-body>oname(set*)</pointcut-body>
  </pointcut>
  <advice name="logSetter" adviceType="before" ref-pointcut="allSetter">
    <advice-body>Log.write()</advice-body>
  </advice>
</aspect>
```

6 Discussion

It is not easy to quantify the effectiveness of AspectM because MDA based on aspect orientation is still young. In this section, we evaluate AspectM qualitatively based on our experience.

We can extend the functionality of the model compiler by adding aspect definitions. However, it is not realistic for a modeler to define all of the aspects needed to construct a model compiler from scratch. It is necessary for model transformation foundations, aspects commonly applied to many transformations, to be pre-defined by model compiler developers. For example, it is preferable to prepare aspect libraries that support de facto standard platforms such as J2EE and .NET. It is also useful to construct aspect libraries that support platform-independent model transformations commonly applied to many applications: fusion of classes having certain kinds of patterns, generation of setter/getter methods, change of naming conventions, and so on. Although aspect libraries are effective, it may be inconvenient to construct them by defining aspect diagrams because the number of aspect definitions tends to be large. It would be more convenient to use XML formats in the case of aspect library development. If a set of aspect libraries could be provided by model compiler developers, modelers would have only to define application-specific aspects as shown in Figure 6.

AspectM includes JPMs supported by major AOP languages. However, it is still not clear whether all kinds of model transformations can be described by the six JPMs. We think that there are situations for which new kinds of JPMs must be introduced. It would be better if a modeler can modify the AspectM metamodel using the model editor. This function can be considered as a modeling-level reflection, a kind of compile-time reflection.

In AspectM, we regard mechanisms explained by extended JPMs as aspect orientation. This definition might be slightly different from that of ordinary aspect orientation. If AspectM is not available, the users cannot describe such a transformation rule within UML since AspectM deals with meta concerns and UML deals with base-level concerns. In AspectJ, for example, a crosscutting concern can be described in Java but the resulting code will be tangled. This means that an aspect in AspectJ is not a meta concern. For this reason, it could be argued that AspectM is not an aspect-oriented language but a meta language for model transformations. However, AspectM can describe not only model transformation rules but also ordinary crosscutting concerns such as logging. AspectM unifies lightweight meta-programming with ordinary aspect orientation by extending the idea of JPMs. It is not necessarily easy to separate platform-specific

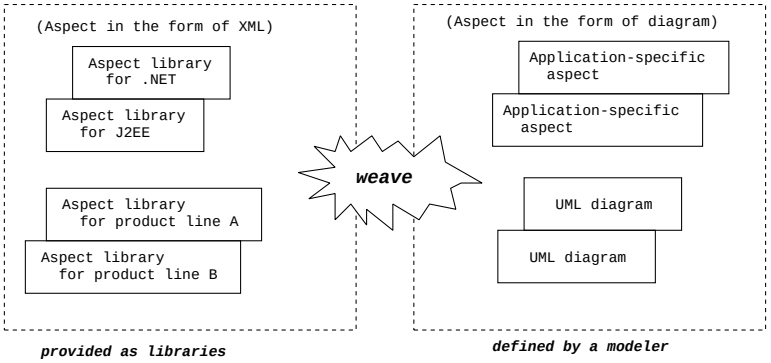


Fig. 6. Software development process using AspectM

concerns with only ordinary aspect orientation. In [18], not only AspectJ but also Java reflection is used for describing database concerns. The approach of AspectM can be considered reasonable.

7 Related Work

There has been research that has attempted to apply aspect-oriented mechanisms in the modeling phase. D.Stein et. al. proposed a method for describing aspects as UML diagrams[21]. In this work, an aspect at the modeling-level was translated into the corresponding aspect at the programming language level, for example an aspect in AspectJ. Y.Han et. al. proposed a meta model and modeling notation for AspectJ[11]. An aspect in AspectM is not mapped to an element of a specific programming language, but operates on UML diagrams. U.Aßmann and A.Ludwig claimed that aspect weaving could be represented as graph rewriting[1]. A UML diagram also can be regarded as graph. J.Sillito et. al. proposed the concept of usecase-level pointcuts, and showed the effectiveness of JPMs in early modeling phases[20]. E.Barra et. al. proposed an approach to an AOSD working method, using the new elements added in UML 2.0[3].

There is a standard model transformation language called QVT (Queries, Views, and Transformations)[17] in which model elements to be transformed are selected by query facilities based on OCL (Object Constraint Language)[23], and are converted using transformation descriptions. Since the purpose of QVT is to describe model transformations, QVT does not provide facilities for describing crosscutting concerns explicitly. AspectM can describe not only model transformation rules but also other kinds of crosscutting concerns.

Domain-specific aspect-oriented extensions are important. Early AOP research aimed at developing programming methodologies in which a system was composed of a set of aspects described by domain-specific AOP languages[12]. Domain-specific extensions are necessary not only at the programming stage but also at the modeling stage. J.Gray provided significant research on topics

including aspect orientation, model-driven developments, and domain-specific languages[8][9]. He proposed a technique of aspect-oriented domain modeling (AODM), and introduced a language called ECL (Embedded Constraint Language), an extension of OCL. ECL included the idea of QVT and provided facilities for adding model elements such as attributes and relations. Although the approach of AODM was similar to AspectM, the purpose of AODM was to realize domain-specific languages. He also proposed an approach that used a program transformation system as the underlying engine for weaver construction[10]. M.Shonle et. al. proposed an extensible domain-specific AOP language called XAspect that adopted plug-in mechanisms[19]. Adding a new plug-in module, we can use a new kind of aspect-oriented facility. CME (Concern Manipulation Environment)[4] adopted an approach similar to XAspect.

AspectM can be considered an XML-based AOP language. There are several AOP languages that can describe aspects in XML formats. AspectWerkz[2] is one such language. However, aspects in AspectWerkz are strongly related to an AspectJ-like JPM, and do not support multiple JPMs as in AspectM. Using AspectM, we can use multiple pieces of design information in describing modeling-level pointcuts. This is one of the advantages of applying aspect orientation to the modeling level. Another approach for enriching pointcuts is to adopt the functional XML query language XQuery[24]. M.Eichberg, M.Mezini, and K.Ostermann investigated the use of XQuery for specification of pointcuts[7].

Introducing AspectM, model transformation rules can be accumulated as reusable software components. This approach is similar to that of Draco[16] proposed by J. Neighbors in 1980s. In Draco, software development processes were considered as a series of transformations: requirements are transformed into analysis specifications; analysis specifications are transformed into design specifications; and design specifications are transformed into source code. These transformations were componentized in Draco. J. Neighbors claimed that software development processes could be automated by composing these transformation components. In AspectM, these components can be described by aspects.

8 Conclusion

We proposed a method for constructing an extensible model compiler based on aspect orientation. A modeler can extend model transformation rules by defining new aspects in the process of modeling. We believe that the idea of AspectM will provide a new research direction for model compiler construction.

References

1. Aßmann, U. and Ludwig, A.: Aspect Weaving as Graph Rewriting, In *Proceedings of Generative Component-based Software Engineering (GCSE)*, pp.24-36, 1999.
2. Aspectwerkz. http://aspectwerkz.codehaus.org/
3. Barra, E., Genova, G., and Llorens, J.: An approach to Aspect Modeling with UML 2.0, The 5th Aspect-Oriented Modeling Workshop, 2004.

4. Concern Manipulation Environment (CME): A Flexible, Extensible, Interoperable Environment for AOSD, http://www.research.ibm.com/cme/.
5. Czarnecki, K., and Eisenecker, U. W.: *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.
6. Early Aspects, http://early-aspects.net/.
7. Eichberg, M., Mezini, M., and Ostermann, K.: Pointcuts as Functional Queries, In *Proceedings of International Conference on Asian Symposium (APLAS 2004)*, pp.366-382, 2004.
8. Gray, J.: Aspect-Oriented Domain-Specific Modeling: A Generative Approach Using a Meta-weaver Framework Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, Vanderbilt University, 2002.
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. Han, Y., Kniesel, G., and Cremers, A., B.: A Meta Model and Modeling Notation for AspectJ, The 5th Aspect-Oriented Modeling Workshop, 2004.
12. 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.
13. 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.
14. 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.
15. MDA, http://www.omg.org/mda/.
16. Neighbors, J.: The Draco Approach to Construction Software from Reusable Components, In *IEEE Transactions on Software Engineering*, vol.SE-10, no.5, pp.564-573, 1984.
17. QVT, http://qvtp.org/.
18. Rashid, A. and Chitchyan, R.: Persistence as an Aspect, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pp.120-129, 2003.
19. Shonle, M., Lieberherr, K., and Shah, A.: XAspects: An Extensible System for Domain-specific Aspect Languages, In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003), Domain-Driven Development papers*, pp.28-37, 2003.
20. Sillito, J., Dutchyn, C., Eisenberg, A.D., and Volder, K.D.: Use Case Level Pointcuts, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2004)*, pp.244-266, 2004.
21. Stein, D., Hanenberg, S., and Unland, R.: A UML-based aspect-oriented design notation for AspectJ, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pp.106-112, 2002.
22. Struts, http://struts.apache.org/.
23. Warmer, J., and Kleppe, A.: *The Object Constraint Language Second Edition —Getting Your Models Ready for MDA*, Addison Wesley, 2003.
24. XQuery, http://www.w3.org/TR/xquery/.