Model Evolution with Aspect-Oriented Mechanisms

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

Tetsuo Tamai
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

Model-based development is a software development method in which models are created before source code is written. Although the method is effective, we have to modify models when we face evolution such as change of platforms. These modifications crosscut over many places in the models, and tend to cause unexpected errors. In order to tackle this problem, we propose a method for model evolution using model transformations based on aspect orientation, a mechanism that modularizes crosscutting concerns. A modeler can extend model transformation rules by defining new aspects in the process of modeling. In this paper, we demonstrate the effectiveness of aspect orientation in terms of model evolution.

1. Introduction

A software development process consists of multiple phases including analysis, design, and implementation: user requirements are refined in an analysis phase; software architecture is determined in a design phase; and program is implemented based on design decisions. Unified modeling language (UML)[19] is used in each phase. For example, use case diagrams are used for extracting functional requirements, and class diagrams are used for representing static software architecture. Although UML-based software developments are effective, it is not necessarily easy to reuse a design model because platform-independent descriptions and platform-specific descriptions are mixed in the same design model. The term *platform* includes OS, middleware such as database systems and application servers, and application development frameworks. When we want to reuse a previous design model, we must conform the model to a new platform by modifying model elements that depend on platform specifications. These modifications crosscut over many places in the model, and tend to cause unexpected errors. Platform-specific concerns are certain kinds of crosscutting concerns.

In order to deal with this problem, OMG (Object Management Group)[13] proposes model-driven architecture (MDA)[11] in which UML 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. It is not necessary to modify a design model for conforming it to a specific platform. We have only to use a model compiler that supports the platform. MDA facilitates model-based agile software development process[2] because we have only to maintain a model. We need not to maintain source code generated from a model.

Although MDA is effective for software development, it mainly focuses on platform-specific concerns. We have to modify models when we face evolution related to other kinds of concerns including application-specific optimization, security policies, and deployment.

This paper proposes a method for dealing with model evolution using model transformations based on aspect orientation[4][8][9], a mechanism that modularizes crosscutting concerns as aspects. Previously we proposed an aspect-oriented modeling language called AspectM (Aspect for modeling) that supports modeling-level aspects[17][18]. Using AspectM, a modeler can extend model transformation rules by defining new aspects in the process of modeling. In this paper, we demonstrate the effectiveness of

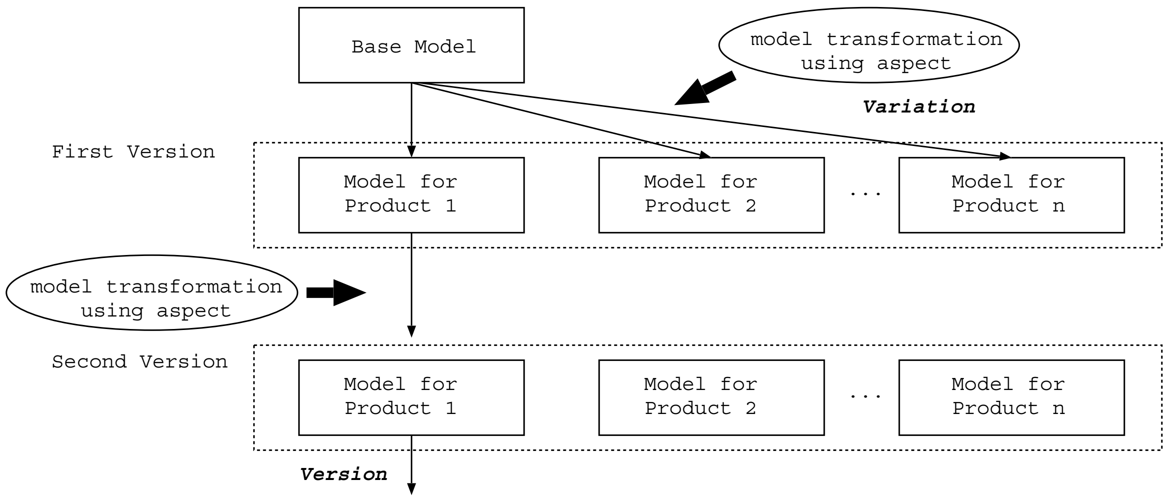


Figure 1. Evolution at the modeling level

AspectM in terms of model evolution.

The remainder of this paper is structured as follows. In Section 2, we illustrate model evolution in terms of transformations. In Section 3, we show a method for model transformations based on aspect orientation, and introduce AspectM for supporting the method. In Section 4, we show a model evolution example using AspectM. In Section 5, we show a relation between evolution and metamodel. In Section 6, we discuss issues on aspect-based model transformations. Lastly, Section 7 concludes this paper.

2. Model evolution

Using model transformations, a series of models can be generated from a single abstract model that embodies only core concerns such as business logics. The model does not include such concerns as platforms and application-specific optimizations. These concerns vary according to product specifications. Model transformations enable us to shift from code-centric product-line engineering (PLE)[3] to model-centric PLE.

In the viewpoint of PLE, there are two kinds of evolution: variation and version as shown in Figure 1. The former indicates a set of products corresponding to a specific version. Using model transformations, we can generate these product variations and product versions. This paper proposes a method for defining model transformation rules using aspect-oriented mechanisms.

3. AspectM

In this section, we briefly excerpt an aspect-oriented modeling language AspectM from the previous work[17][18], and then illustrate model transformations using aspects.

3.1. Aspect orientation at the modeling level

Aspect-oriented programming (AOP) is based on the join point model (JPM) consisting of join points, pointcuts, and advice[9]. 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.

Although JPMs have been proposed as a mechanism at the programming level, they can be applied for constructing a model compiler that transforms a UML model into another UML model. Figure 2 shows an example of a model transformation. In Figure 2, 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`.

AspectM supports six kinds of modeling-level JPMs: PA (pointcut & advice as in AspectJ[1][9]), CM (composition as in Hyper/J[16]), NE (new element), OC (open class as in AspectJ inter-type declaration), RN (rename), and RL (relation). Figure 2 is an example of an OC. Table 1 shows the outline of these JPMs.

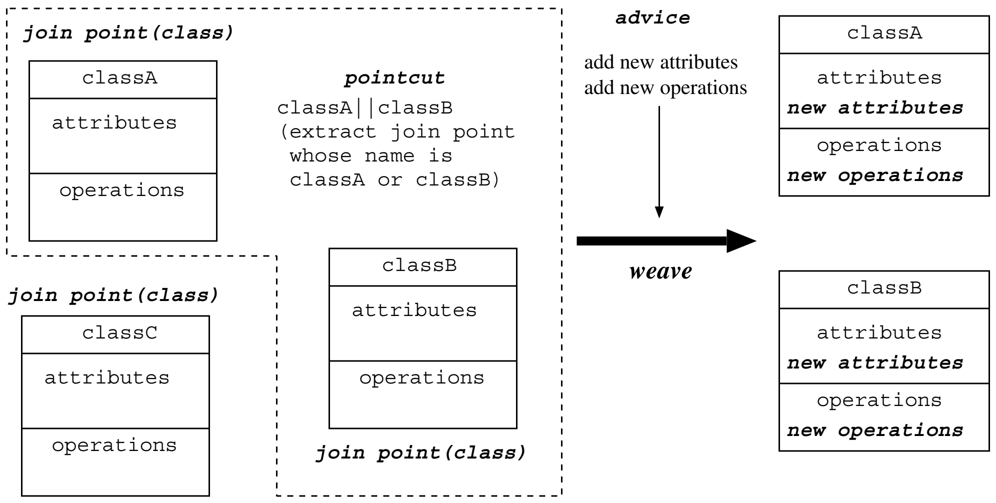


Figure 2. Aspect orientation at the modeling level

JPM type	Join point type	Advice type
PA (pointcut & advice)	operation	before, after, around
CM (composition)	class	merge-by-name
NE (new element)	class diagram	add-class, delete-class
OC (open class)	class	add-operation, delete-operation add-attribute, delete-attribute
RN (rename)	class, operation, attribute	rename
RL (relation)	class	add-inheritance, delete-inheritance add-aggregation, delete-aggregation add-relationship, delete-relationship

Table 1. JPMs in AspectM

3.2. Language features

AspectM provides a set of notations for describing aspects. In AspectM, an aspect can be described in either a diagram or an XML (eXtensible Markup Language) format.

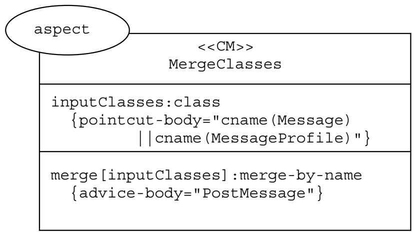
Figure 3 shows an example of the AspectM diagram notations and the corresponding XML formats. AspectM is not only a diagram language but also an XML-based AOP language. AspectM provides the two kinds of aspects: an ordinary aspect and a component aspect. A component aspect is a special aspect used for composing aspects. 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.

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 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). 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. An advice is applied at join points selected by a pointcut.

3.3. Metamodel for aspects

AspectM is defined as an extension of the UML metamodel[1]. OMG defines the four-level metamodel hierarchy consisting of M0, M1, M2, and M3: M0 contains the data

[1]Currently, AspectM is based on UML 1.5. We plan to support UML 2.0 in the near future.


aspect
<<CM>>
MergeClasses
inputClasses:class
{pointcut-body="cname(Message)
||cname(MessageProfile)"}
merge[inputClasses]:merge-by-name
{advice-body="PostMessage"}

```
<aspect name="MergeClasses"
        type="ordinary" jpm="CM">
  <pointcut name="inputClasses"
            type="class">
    <pointcut-body>
      cname(Message)||cname(MessageProfile)
    </pointcut-body>
  </pointcut>
  <advice name="merge" type="merge-by-name"
          ref-pointcut="inputClasses">
    <advice-body>PostMessage</advice-body>
  </advice>
</aspect>
```

Figure 3. AspectM notation

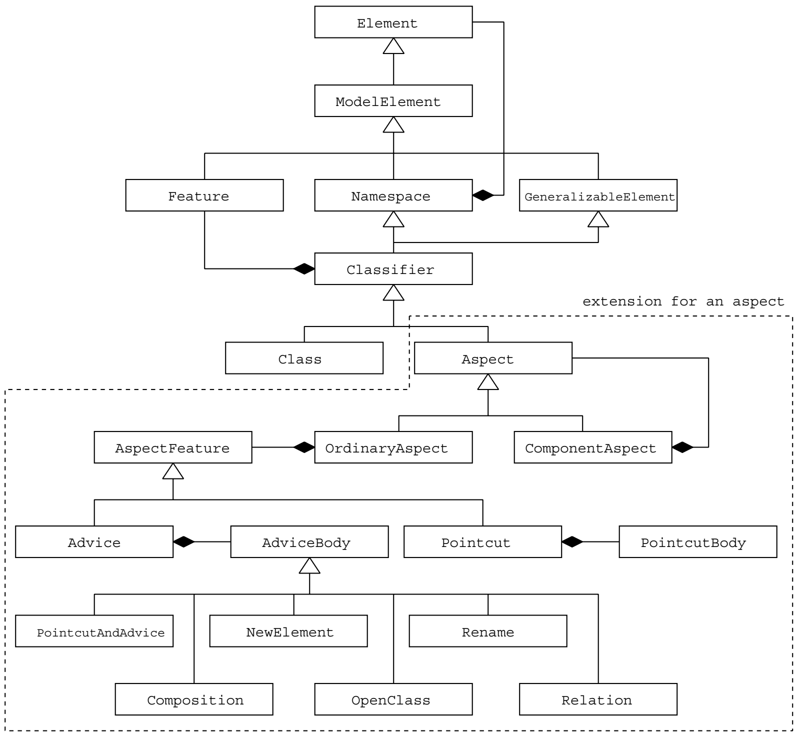


Figure 4. AspectM metamodel

such as object instances; M1 contains application models described by a modeling language such as UML; M2 contains metamodels that defines the modeling language; M3 contains metametamodels that defines metamodels in M2. For example, a class diagram and the UML metamodel reside at the level M1 and M2, respectively. MOF (Meta Object Facility) proposed by OMG is a language for modeling a metamodel. MOF resides at the level M3. Figure 4 shows a part of our metamodel (parameterization is not included). This metamodel resides at the level M2. Introducing this metamodel, we can define an aspect in a UML diagram that resides at the M1 level.

Now, we explain the AspectM metamodel briefly. The `OrdinaryAspect` class and the `ComponentAspect` represent an ordinary aspect and a component aspect, respectively. The `Aspect` class, which inherits the `Classifier` class, is a super class of the above two classes. The GoF Compositor pattern[6] is applied to define a component aspect. An aspect consists of pointcuts and advice. They are represented by the `Pointcut` class and the `Advice` class, respectively. The `AspectFeature` class is a super class of these two classes, and has an aggregation relation with the `OrdinaryAspect` class. The `Pointcut` class and the `Advice` class have the `PointcutBody` class and the `AdviceBody` class, respectively. Concrete advice bodies corresponding to the six JPMs are defined as subclasses of the `AdviceBody` class: `PointcutAndAdvice` for PA, `Composition` for CM, `NewElement` for NE, `OpenClass` for OC, `Rename` for RN, and `Relation` for RL. The `PointcutBody`

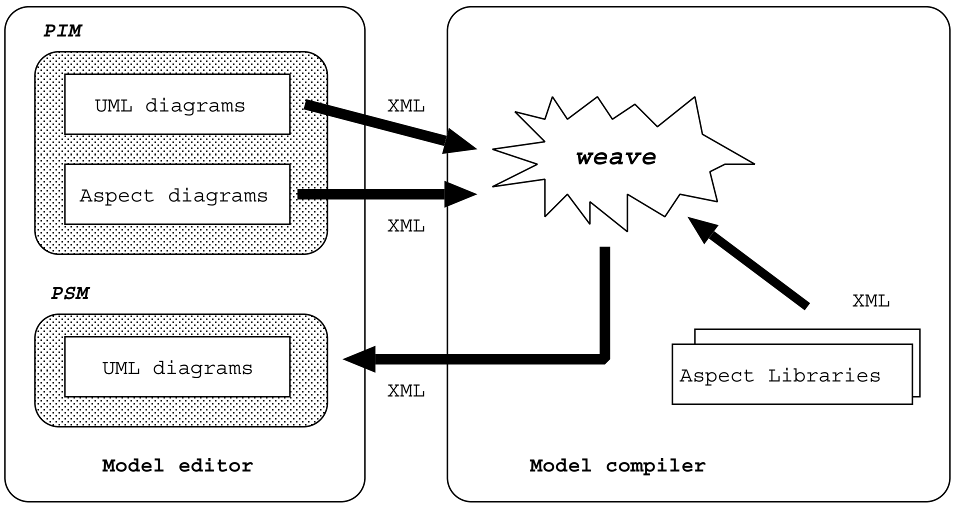


Figure 5. AspectM implementation outline

is common to all JPMs because pointcuts can be specified uniformly. We can change/add/delete JPMs by changing/adding/deleting these subclasses.

3.4. Supporting tool

We have developed a prototype of AspectM. The tool for supporting AspectM consists of the model editor and the model compiler as illustrated in Figure 5[2].

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. We defined a metamodel for aspects, and generated the model editor from the metamodel using EMF. 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.

4. Model evolution using AspectM

We illustrate model evolution using the following simple bulletin board system as an example[3]: *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[15].

[2]A PSM class can be translated into Java source code although this is not shown in the figure.

[3]This is the same example used in our previous work[17][18]. In this paper, we demonstrate the effectiveness of AspectM from the viewpoint of model evolution.

We illustrate an evolution process consisting of the following two versions and one variation: 1) make a product that is executable on the Struts platform (the first version); 2) optimize memory usage (the second version); and 3) add a logging function for debugging (variation of the second version).

4.1. Original model

We define PIMs that do not depend on a specific product. There are two PIMs in this example as shown in the left of Figure 6[4]. 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 a message id and a date, is a PIM defined from the viewpoint of the system administrator.

4.2. The first version

As the first version, we generate a product that is executable on Struts. We transform the PIMs (the above original model) into a PSM targeted to Struts. Figure 6 illustrates this transformation process. The following shows the transformation steps from the PIMs into the PSM .

4.2.1 Step 1

First, we deal with the above two viewpoints. The two PIM classes `Message` and `MessageProfile` are merged into the single class `PostMessage`.

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

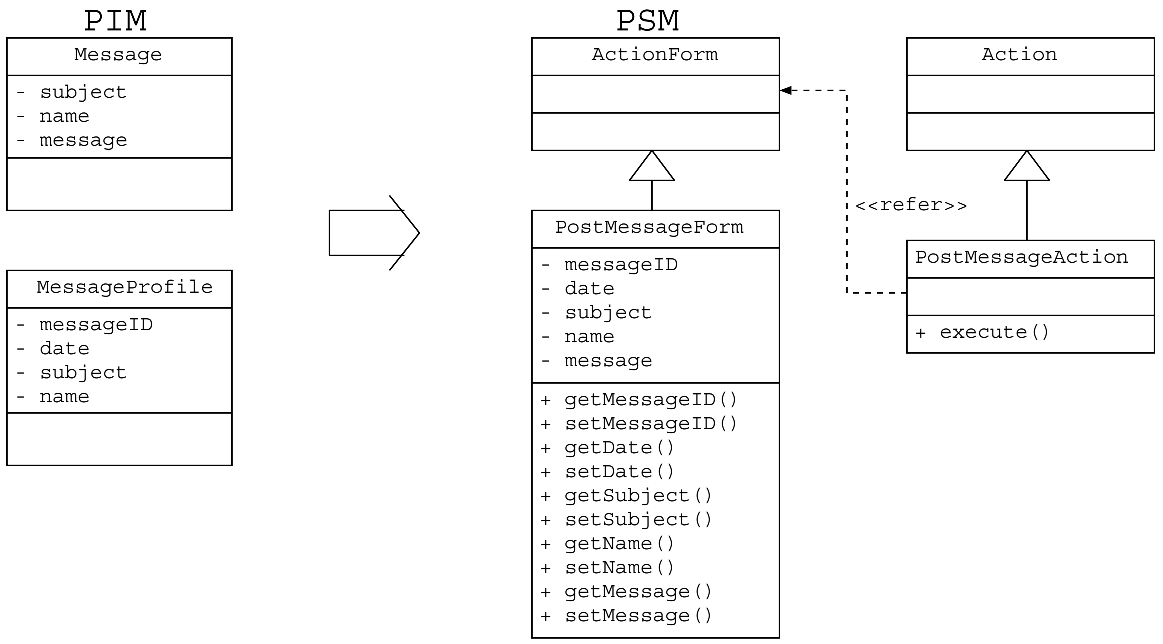


Figure 6. Model transformation for Struts

4.2.2 Step 2

In Step 2 and Step 3, we conform the merged class to the conventions specified in Struts. In Struts, a request from a web browser is stored in an object called action form bean. The `PostMessage` class is transformed to an action form bean class. First, the name of the `PostMessage` class is changed to `PostMessageForm`. Next, the parent class of the `PostMessageForm` is set to the `ActionForm` framework class. After that, a set of accessors (setter/getter) are added to the `PostMessageForm` class.

4.2.3 Step 3

In Struts, an action logic, which 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 `PostMessageAction` class.

Figure 3 shows Step 1 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. Step 2 and Step 3 can be also realized with the same approach as shown in Table 2.

4.3. The second version

As the second version, we optimize the first version in terms of memory usages. Adopting AspectM, we can extend the functionality of the model transformations by adding aspect definitions. The following is the aspect that deletes the `date` attribute when the two classes `Message` and `MessageProfile` are merged. An XML format is used here due to limitations of space.

```
<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 PLE. 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

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

applied after the `MergeClasses` aspect is applied. Using AspectM, a process of tuning up can be componentized as an aspect.

4.4. Variation of the second version

As the variation of the second version, we add a logging function. This variation is necessary in the case of debugging or testing. The logging function may be eliminated in the final product. The following is an aspect for logging setter method calls. The `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>
```

5. Evolution of metamodel

AspectM is effective for model evolution because a modeler can extend the functionality of the AspectM model compiler by defining new aspects as shown in Section 4. However, the ability of model evolution is restricted to the six JPMs. A modeler cannot define new kinds of transformation rules that need new JPMs. There may be questions: are they enough for dealing with all kinds of model transformations?, and is there a method for adding new JPMs or modifying existing JPMs? AspectM includes JPMs supported by major AOP languages. However, it is not still 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. In the AspectM metamodel, we can introduce a new kind of advice by defining a new subclass of the `AdviceBody` class. In this situation, model transformations can be categorized as follows: transformation using pre-defined aspect libraries; transformation using user-defined aspects based on pre-defined metamodels; and transformation using user-defined aspects based on user-defined metamodels. This function can be considered as a modeling-level reflection, a kind of compile-time reflection. The idea of extensible programming languages, such as computational reflection[14][10] and metaobject protocols[7] would be also useful at the modeling-level. We think that metaobject protocols in AspectM can be provided as hot-spots such as the `AdviceBody` class. Designs of metaobject protocols are similar to designs of an application framework in which hot-spots should be exposed. We think that the idea of the modeling-level reflection will be important research issues in the future.

6. Discussion

AOP is effective in unanticipated software evolution because crosscutting concerns can be added or removed without making invasive modifications on original programs. This is also true at the modeling-level aspect orientation as shown in this paper. That is, we can deal with model evolution by adding aspect definitions. However, it is not realistic for a modeler to define all of aspects in terms of scalability. 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 including J2EE, Web service, and .NET. If a set of aspect libraries are provided by model compiler developers, modelers have only to define application-specific aspects. As shown in Section 3.2, AspectM supports generic aspects that is effective for developing libraries. By filling parameters, a specific aspect is generated from a common generic aspect.

Introducing AspectM, model transformations, which is represented as an aspect, can be accumulated as aspect libraries. This approach is similar to that of Draco[12] 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; design specifications are transformed into source code. These transformations are 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. Although Draco's approach failed to become popular, the idea of Draco is effective even now.

In order to use AspectM for model transformation, it is important to have good original models. Otherwise, we need to check the all the aspects in the library for their numerous combinations. There can be consistency problem on the interactions among aspects, which may become another source of errors. This problem is essential for model evolution. To deal with this problem, we plan to develop a mechanisms for verifying a model compiler that consists of pre-defined aspects and user-defined aspects.

7. Conclusion

In this paper, we showed model evolution in terms of transformations based on aspect-oriented mechanisms. A modeler can extend model transformation rules by defining new aspects when evolution occurs. The idea proposed in this paper will give a new research direction in terms of evolution on MDA.

References

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

[2] Cockburn, A.: *Agile Software Development*, Addison-Wesley, 2000.

[3] Czarnecki, K., and Eisenecker, U. W.: *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.

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

[5] EMF, http://www.eclipse.org/emf/.

[6] Gamma, E. et al.: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

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

[8] 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.

[9] 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.

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

[11] MDA, http://www.omg.org/mda/.

[12] 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.

[13] OMG, http://www.omg.org/.

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

[15] Struts, http://struts.apache.org/.

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

[17] Ubayashi, N. and Tamai, T.: Concern Management for Constructing Model Compilers, In *Proceedings of First International Workshop on the Modeling and Analysis of Concerns in Software (MACS 2005) (Workshop at ICSE 2005)*, pp.9-13, 2005.

[18] Ubayashi, N., Tamai, T., Sano, S., Maeno, Y., and Murakami, S.: Model Compiler Construction Based on Aspect-Oriented Mechanisms, In *Proceedings of Fourth International Conference on Generative Programming and Component Engineering (GPCE 2005)*, to appear.

[19] UML, http://www.uml.org/.