A Reflective Aspect-oriented Model Editor Based on Metamodel Extension

Naoyasu Ubayashi, Shinji Sano and Genya Otsubo
Kyushu Institute of Technology, Japan
ubayashi@acm.org, {sano,otsubo}@minnie.ai.kyutech.ac.jp

Abstract

AspectM, an aspect-oriented modeling language, provides not only basic modeling constructs but also an extension mechanism called metamodel access protocol (MMAP) that allows a modeler to modify the metamodel. MMAP consists of metamodel extension points, extension operations, and primitive predicates for defining pointcut designators. In this paper, a reflective model editor for supporting MMAP is proposed. A new modeling construct can be introduced by extending the metamodel. This mechanism, a kind of edit-time structural reflection, enables a modeler to represent domain-specific crosscutting concerns.

1 Introduction

Aspect-oriented programming (AOP) [9] 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) 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 weaver inserts advice code at the join points selected by pointcut designators.

Aspect orientation has been proposed for coping with concerns not only at the programming stage but also at the early stages of the development phases such as requirements analysis and architecture design. Aspects at the modeling-level are typically static structures, such as UML diagrams, and are not much concerned with behavior as in AOP.

We previously proposed a UML-based aspect-oriented modeling language called AspectM [15]. 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 model compiler.

AspectM provides not only major JPMs but also a mechanism called metamodel access protocol (MMAP) [16] that allows a modeler to modify the metamodel, an extension of the UML metamodel. The mechanism enables a modeler to define a new JPM that includes domain-specific join points, pointcut designators, and advice. There are two approaches for extending model elements in UML: a lightweight approach using UML profiles and a heavyweight approach that extends the UML metamodel by using MOF (Meta Object Facility). While it is easy to adopt UML profiles, there are applications that need metamodel extension because stereotypes in UML profiles are insufficient: the typing of tags is weak; and new associations among UML metamodel elements cannot be declared [3]. 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 a modeler to extend the UML metamodel by using the full power of the MOF. MMAP aims at a middleweight approach that restricts available extension by MOF. Although the notion of MMAP is useful, it needs tool support.

In this paper, a reflective model editor for supporting MMAP is proposed. This reflective mechanism, a kind of edit-time structural reflection, enables a modeler to define a JPM specific to a system or a family of systems. For example, a modeler can define an aspect that captures a group of methods that are targets of a domain-specific logging.

The remainder of the paper is structured as follows. Section 2 explains AspectM and MMAP. In Section 3, the concept of a reflective model editor is introduced. Section 4 shows an implementation method. Section 5 introduces related work. Concluding remarks are provided in Section 6.

2 AspectM and MMAP

This section briefly excerpts the overview of AspectM and MMAP from our previous work [15, 16].

2.1 Aspect orientation at the modeling-level

Although JPMs have been proposed as a mechanism at the programming-level, they can be applied to the

Table 1. JPM in AspectM

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

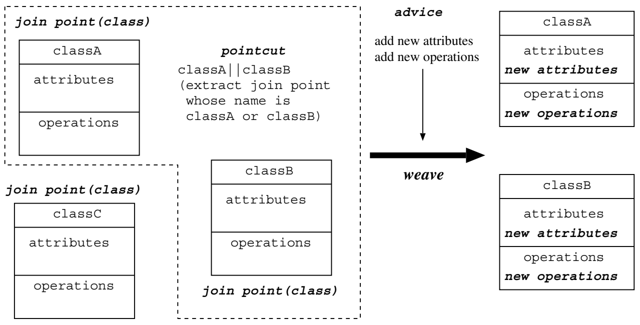


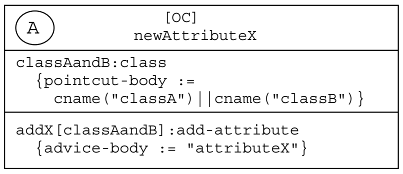
Figure 1. Modeling-level aspect orientation

modeling-level. Figure 1 shows an example of the modeling-level aspects. 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 class A, 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. The effect of the advice crosscuts these two classes. As shown here, aspects at the modeling-level are typically static structures, and are not much concerned with behavior.

AspectM supports six kinds of modeling-level JPMs: PA (pointcut & advice as in AspectJ [10]), CM (composition as in Hyper/J [14]), 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.

2.2 AspectM language features

Figure 2 shows an aspect diagram that corresponds to Figure 1. The notation is similar to that of the UML class diagram. An aspect diagram is separated into three compartments: aspect name and JPM type, pointcut definitions, and advice definitions. An aspect name and a JPM type are described in the first compartment. Pointcut definitions are described in the second compartment. Each of them consists of a pointcut name, a join point type, and a pointcut



```
A              [OC]
           newAttributeX

classAandB:class
  {pointcut-body :=
    cname("classA")||cname("classB")}

addX[classAandB]:add-attribute
  {advice-body := "attributeX"}
```

Figure 2. AspectM notation

body. In pointcut definitions, we can use designators including cname (class name matching), aname (attribute name matching), and oname operation name matching). These pointcut designators can be defined using MMAP as explained in 2.3. 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.

2.3 MMAP

Figure 3 shows a part of the AspectM metamodel defined as an extension of the UML metamodel. The Aspect class inherits the Classifier class. Pointcuts and advice are represented by the Pointcut class and the Advice 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 class is common to all JPMs because pointcuts can be specified uniformly.

MMAP, a set of protocols exposed for a modeler to access the AspectM metamodel, 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. The extension points includes Class, Attribute, Operation, and a set of JPM metaclasses. In Figure 3, a class with a thick line is an extension point. An extension operation is a modeling activity allowed at the exposed extension points. There are three operations including *define subclasses*, *add attributes to subclasses*, and *create associations among subclasses*. Table 2 is a list of primitive predicates. Using these predicates, pointcut designators can be defined as below. The defined pointcut designator represents all elements that satisfy the right-hand side predicates.

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

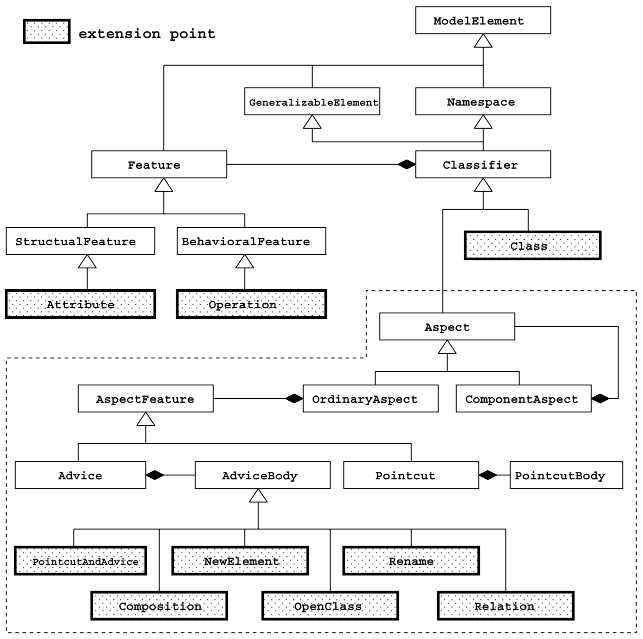


Figure 3. AspectM metamodel

Table 2. Primitive predicates

Predicate	Explanation
meta-class-of(mc, c)	*mc* is a metaclass of *c*
member-of(m, c)	*m* is a member of a class *c*
value-of(v, a)	*v* is value of an attribute *a*
super-class-of(c1, c2)	*c1* is a superclass of *c2*
related-to(c1, c2)	*c1* is related to *c2*

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

The idea of MMAP originates in the mechanisms of extensible programming languages, such as metaobject protocol(MOP) [8] and computational reflection [11] in which interactions between the base-level (the level to execute applications) and the meta-level (the level to control meta information) are described in the same program. There are two kinds of reflection: behavioral reflection and structural reflection. MMAP corresponds to the latter. That is, MMAP focuses on the reflection whose target is a model structure.

3 Reflective model editor

We have developed a prototype of the reflective model editor for supporting MMAP. Our previous editor [15] did not support a reflective mechanism. In this section, the concept of the reflective model editor and the metamodel extension procedures using this editor are demonstrated.

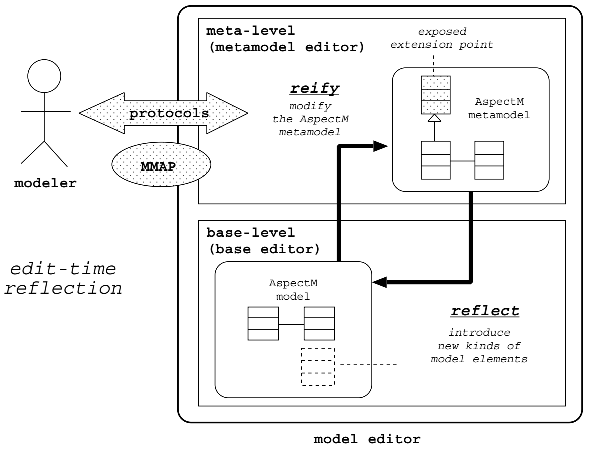


Figure 4. Reflection mechanism

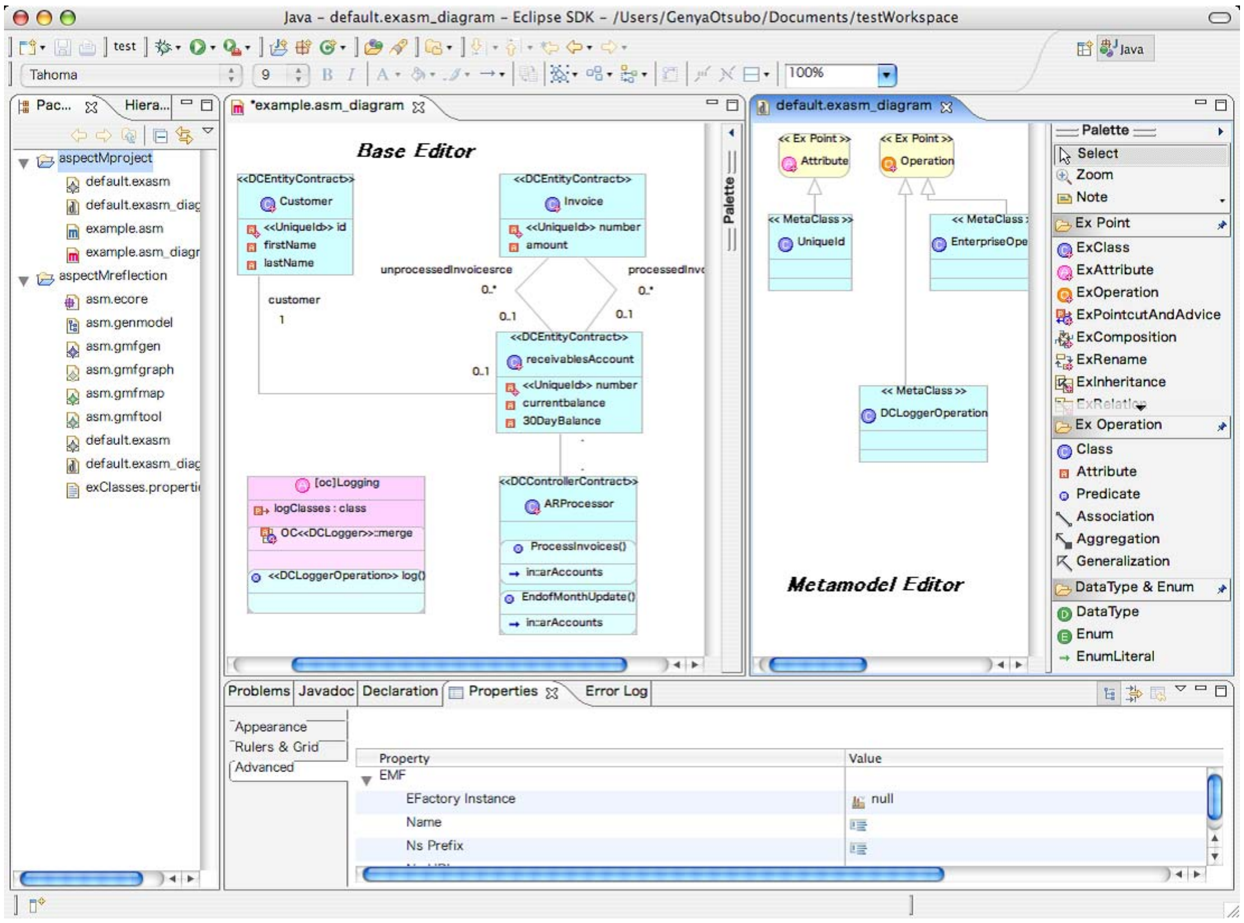
3.1 Concept

The reflective model editor provides not only facilities for editing UML and aspect diagrams but also a mechanism for structural reflection based on MMAP. This editor allows a modeler to not only edit application models but also extend, modify, and customize the AspectM metamodel.

Figure 4 illustrates the concept of the editor consisting of two parts: the base editor and the metamodel editor. The former is the editor for base-level modeling, and the latter is the editor for modifying the AspectM metamodel and defining pointcut designators using primitive predicates. Figure 5 shows the user interface of the reflective model editor. The metamodel editor exposes extension points. Only extension points are displayed on the editor screen. Other metamodel elements are not visible to a modeler, and not allowed to be modified. At an extension point, an extension operation such as *define subclasses* can be executed. This extension operation corresponds to *reification* in computational refection. The result of extension operations enhances the functionality of the base editor. That is, new kinds of model elements can be used in the base editor. This corresponds to the *reflect* concept in computational relection. In reflective programming, a programmer can introduce new language features using MOP. In aspect modeling, a modeler can introduce new model elements using MMAP.

3.2 Extension procedure

Using an example, we illustrate the extension procedure. Figure 6 is a model of an invoice processing system (the class diagram is cited from [3]). This model includes several domain-specific model elements that cannot be described by only using the original AspectM facility. This model is comprised of two kinds of domain-specific distributed components: `DCEntityContract`



Property	Value
EMF	
EFactory Instance	null
Name	
Ns Prefix	

Figure 5. Reflective model editor

for defining the contract of a distributed entity component and DCControllerContract for defining the contract of a distributed controller component. The Logging aspect adds a log operation to the DCEntityContracts components whose UniqueId is not assigned by users. Although the bodies of the logClasses pointcut and the merge advice whose type is OC<<DCLogger>> are invisible in Figure 6, these bodies are defined as below. The Logging aspect includes domain-specific pointcut designators and advice. The DCEntityContract element, which is regarded as one of the domain-specific join points for a family of invoice processing systems, is the target of the DCEntityContract_UniqueId_isUserAssigned pointcut. The functionality of the OC<<DCLogger>> advice is restricted because only the operations with <<DCLoggerOperation>> are the target of *add/delete-operation in OC*.

```
pointcut-body:=
  !DCEntityContract_UniqueId_isUserAssigned(*)
advice-body:=<<DCLoggerOperation>>log()
```

Steps for modeling-level reflection

Using the reflective model editor, a modeler can edit the model shown in Figure 6. The following is the outline of extension steps: 1) execute extension operations; 2) assign a graphic notation to a new model element; 3) check the consistency between the previous metamodel and the new metamodel; 4) regenerate the AspectM metamodel; and 5) restart the base editor.

In step 1, extension operations are executed at exposed extension points in order to introduce new domain-specific model elements as shown in Figure 7. The constraints among new model elements can be specified using OCL (Object Constraint Language). The model elements that violate the OCL descriptions can be detected by the editor. Pointcut designators are also defined as below. 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 6, the negation of this pointcut designator selects the two classes Customer and Invoice.

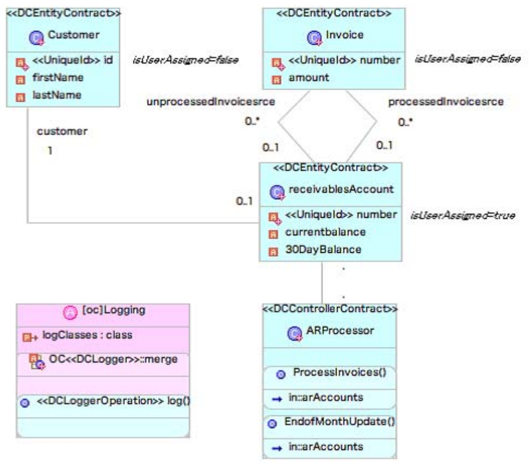


Figure 6. Example of a domain-specific model

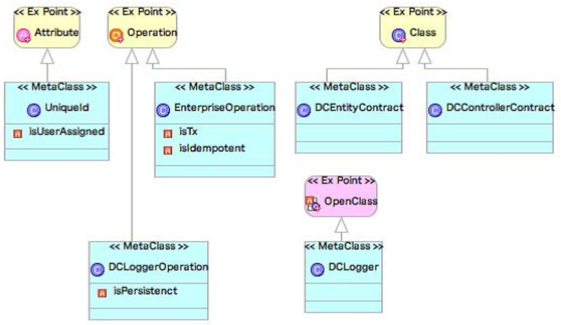


Figure 7. Metamodel extension

```
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")
```

After step 2 – 5, the new model element can be used in the base editor. In the reflective model editor, an extension model is separated from the original AspectM metamodel. Extension models can be accumulated as plug-in components for domain-specific modeling.

4 Implementation

The AspectM tool consists of the reflective model editor and the model compiler that translates a model into Java.

The reflective model editor[1], a plug-in module for Eclipse, is developed using the Eclipse Modeling Framework (EMF) [2] and Graphical Modeling Framework (GMF) [4]. The former is a tool that generates a model editor from a metamodel, and the latter provides a generative component and runtime infrastructure for developing a graphical editor based on EMF. EMF consists of *core EMF*, *EMF.Edit*, and *EMF.Codegen*: the *core EMF* provides a meta model (Ecore) for describing models and runtime support; *EMF.Edit* provides reusable classes for building editors; and *EMF.Codegen* generate code needed to build a complete editor for an EMF model. Since an editor generated from EMF does not provide graphical facilities, GMF is used for this purpose.

The reflective mechanism is implemented as follows: 1) the original AspectM metamodel is defined as an EMF model, and the original base editor is generated using *EMF.Codegen*; 2) the metamodel extension specified by a modeler is saved as an EMF model, and the editor code for the extension is generated using *EMF.Codegen*; and 3) a new plug-in is generated from the code for the base editor and the extension, and replaced with the original plug-in.

In the model compiler that supports MMAP, the metamodel and application models are transformed to a set of Prolog predicates. For example, the `Invoice` class and related metamodel elements are represented as follows.

```
-- from Invoce class
  meta-class-of("DCEntityContract", "Invoice"),
  member-of("number", "Invoice"),
  meta-class-of("UniqueId", "number"),
  value-of("true", "isUserAssigned").
-- from AspectM metamodel
  member-of("isUserAssigned", "UniqueId").
```

The model compiler transforms a domain-specific pointcut into a Prolog query, and checks whether the query satisfies the above facts. For example, the negation of the `DCEntityContract_UniqueId_isUserAssigned` pointcut selects `Customer` and `Invoice` as join points. The model compiler executes advice at these join points. JPL, a set of Java classes providing an interface between Java and Prolog, provided by SWI-Prolog [13] is used for bridging the model compiler and the Prolog interpretor.

In MMAP, `Advice/AdviceBody` are not exposed as extension points because this extension need new weaver modules that can handle new kinds of advice. Adopting our approach, the model compiler need not be modified even if the metamodel is modified by the reflective model editor.

5 Discussion and Related Work

The notion of domain-specific aspect orientation is important. Early AOP research aimed at developing methodologies in which a system was composed of a set of aspects described by domain-specific AOP languages [9]. Domain-specific aspect orientation is necessary not only at the programming stage but also at the modeling stage. J.Gray

[1]The prototype of the refective model editor can be downloaded from http://posl.minnie.ai.kyutech.ac.jp/.

proposed a technique of aspect-oriented domain modeling (AODM) [5] that adopted the Generic Modeling Environment (GME), a meta-configurable modeling framework. The GME provides meta-modeling capabilities that can be adapted from meta-level specifications for describing domains. The GME approach is heavyweight because meta-level specifications can be described fully. On the other hand, our approach is middleweight. Although all of the AspectM metamodel cannot be extended, domain-specific model elements can be introduced at relatively low cost.

To evaluate the effectiveness of our approach, we applied the reflective model editor to a real system. Using the reflective mechanism, we could construct a model editor for describing embedded systems composed of software modules, sensors, actuators, and usage contexts. These model elements and OCL constraints among them could be introduced easily. We did not need to modify the metaclasses that were not exposed by MMAP. From the current our experience, MMAP is sufficient to extend the AspectM metamodel. We believe that the MOP approach in the modeling-level is more effective than the full metamodel extension approaches in terms of the cost and usability.

MMAP is similar to an edit-time metaobject protocol (ETMOP) [1] 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. An ETMOP programmer can define special metaclasses that customize a display and editing. They implemented a REMO for editing UML state charts. Although their research goal that is to provide mechanisms for making programs more visually expressive is similar to our goal, we focus on the provision of middleweight mechanisms for domain-specific expressiveness.

K.Gybels and J.Brichau proposed pattern-based pointcut constructs using logic programming facilities [6]. K.Ostermann, et al. also proposed a pointcut mechanism based on logic queries written in Prolog [12]. The mechanism is implemented for a typed AOP language, called ALPHA. Although our pointcut definition method using Prolog is basically the same with their approaches, the target of our approach is not programming but modeling in which rich pointcuts can be defined because the information of a model is richer than that of a program.

Our approach is effective for software product line (SPL) [7] in which a product is constructed by assembling core assets, components reused in a family of products. These core assets are identified by analyzing features in a family of products. The core assets will be developed efficiently if their specifications can be described using notations specific to the product family. These notations can be introduced easily by using the reflective model editor. Moreover, the extension descriptions can be accumulated as core assets.

6 Conclusion

This paper proposed a reflective model editor. Although the current editor might need refining, it is an important step towards extensible aspect-oriented modeling.

References

[1] Eisenberg, A. and Kiczales, G.: A Simple Edit-Time Metaobject Protocol, *Workshop on Open and Dynamic Aspect Languages (OAL)* 2006.

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

[3] Frankel, D. S.: *Model Driven Architecture*, John Wiley & Sons, Inc., 2003.

[4] GMF, http://www.eclipse.org/gmf/.

[5] Gray, J., et al.: 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.

[6] Gybels, K. and Brichau, J., Arranging Language Features for More Robust Pattern-based Crosscuts, In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pp.60-69, 2003.

[7] Kang, K. C., et al.: Feature-Oriented Product Line Engineering, *IEEE Software*, Vol. 9, No. 4, pp.58-65, 2002.

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

[9] Kiczales, G., et al.: Aspect-Oriented Programming, In *Proceeding of European Conference on Object-Oriented Programming (ECOOP'97)*, pp.220-242, 1997.

[10] Kiczales, G., et al.: An Overview of AspectJ, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2001)*, pp.327-353, 2001.

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

[12] Ostermann, K., Mezini, M., and Bockisch, C.: Expressive Pointcuts for Increased Modularity, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2005)*, pp.214-240, 2005.

[13] SWI-Prolog, http://www.swi-prolog.org/.

[14] 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.

[15] Ubayashi, N., Tamai, T., Sano, S., Maeno, Y., and Murakami, S.: Model Compiler Construction Based on Aspect-Oriented Mechanisms, In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE 2005)*, pp.109-124, 2005.

[16] Ubayashi, N. Tamai, T., Sano, S., Maeno, Y., and Murakami, S.: Metamodel Access Protocols for Extensible Aspect-Oriented Modeling, In *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE 2006)*, pp.4-10, 2006.