An Extensible Aspect-Oriented Modeling Environment for Constructing Domain-Specific Languages*

Naoyasu UBAYASHI† and Yasutaka KAMEI*, Members

SUMMARY AspectM, an aspect-oriented modeling (AOM) 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 navigating the metamodel. Although the notion of MMAP is useful, it needs tool support. This paper proposes a method for implementing a MMAP-based AspectM support tool. It consists of model editor, model verifier, and model verifier. We introduce the notion of edit-time structural reflection and extensible model weaving. Using these mechanisms, a modeler can easily construct domain-specific languages (DSLs). We show a case study using the AspectM support tool and discuss the effectiveness of the extension mechanism provided by MMAP. As a case study, we show a UML-based DSL for describing the external contexts of embedded systems.

key words: aspect-oriented modeling, extensible weaver, DSL

1. Introduction

Aspect-oriented programming (AOP) [20] can separate crosscutting concerns from primary concerns. In major AOP languages such as AspectJ [21], 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 such as requirements analysis and architecture design.

We previously proposed a UML-based aspect-oriented modeling (AOM) language called AspectM [33] that provides not only major JPMs but also a mechanism called metamodel access protocol (MMAP) [34] for allowing 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. Although the notion of MMAP is useful, it needs tool support.

This paper proposes a method for implementing a MMAP-based AspectM support tool. It consists of model editor, model verifier, and model verifier. These tool components provide an extensible AOM environment. We introduce the notion of edit-time structural reflection and extensible model weaving to support MMAP. The model editor supporting 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. A newly introduced JPM can be dealt with by the extensible model verifier supporting MMAP. Although the AspectM language features can be extended by MMAP, it is not necessarily easy to confirm the correctness of the model weaving. The verification such as the model consistency and the aspect interference becomes difficult, because a weaver must be verified whenever AspectM is extended. If a verification mechanism is not provided, it is difficult to confirm whether an extended part does not interfere with other existing parts. If the extension includes defects, models are not woven properly. It is not easy for a modeler to know whether either the original model or the extension is incorrect. To deal with this problem, we propose a verification mechanism for the weaving in extensible AOM languages.

The remainder of the paper is structured as follows. Section 2 explains AspectM briefly, and claims why extension mechanisms are needed. In Sect. 3, the overview of MMAP is illustrated. In Sect. 4, the outline of AspectM support tool is shown. The mechanisms of edit-time reflection, extensible model weaving, and model verification are shown in Sect. 5, 6, 7, respectively. In Sect. 8, we show a case study. Section 9 introduces related work. Concluding remarks are provided in Sect. 10.

2. Motivation

We show the overview of AspectM. The necessity of extensible AOM is also demonstrated.

2.1 Core AspectM

2.1.1 Modeling-Level Aspect Orientation

The language features illustrated in this section do not in-
UBAYASHI and KAMEI: AN EXTENSIBLE ASPECT-ORIENTED MODELING ENVIRONMENT FOR CONSTRUCTING DOMAIN-SPECIFIC LANGUAGES

Fig. 1 Modeling-level aspect orientation.

Table 1 JPM in Core AspectM.

<table>
<thead>
<tr>
<th>JPM</th>
<th>Join point</th>
<th>Advice</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
<td>operation</td>
<td>before, after, around</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add/delete-precondition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add/delete-postcondition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>replace-precondition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>replace-postcondition</td>
</tr>
<tr>
<td>CM</td>
<td>class</td>
<td>merge-by-name</td>
</tr>
<tr>
<td>NE</td>
<td>class diagram</td>
<td>add/delete-class</td>
</tr>
<tr>
<td>OC</td>
<td>class</td>
<td>add/delete-operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add/delete-attribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add/delete-invariant</td>
</tr>
<tr>
<td></td>
<td></td>
<td>replace-invariant</td>
</tr>
<tr>
<td>RN</td>
<td>class, attribute, operation</td>
<td>rename</td>
</tr>
<tr>
<td>RL</td>
<td>class</td>
<td>add/delete-aggregation</td>
</tr>
<tr>
<td>IH</td>
<td>class</td>
<td>add/delete-relationship</td>
</tr>
</tbody>
</table>

include extension mechanisms. We call this language Core AspectM. 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 Fig. 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 Fig. 1, new attributes and operations are added to the two classes classA and classB.

Core AspectM provides seven kinds of modeling-level JPMs as shown in Table 1: PA (pointcut & advice as in AspectJ), CM (class composition), NE (new element), OC (open class as in AspectJ inter-type declaration), RN (rename), RL (relation), and IH (inheritance).

- PA: A join point is an operation. Advice changes the behavior, precondition, or postcondition at join points selected by a pointcut. For the behavior, three kinds of advice can be described: before (a pre-process is added), after (a post-process is added), and around (a process is replaced). Pre-/postconditions of an operation can be added, deleted, and replaced. In the case of addition, a new condition is added by logical operator.
- CM: 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 classes to a single class.
- NE: This 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.
- OC: This 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. Invariants of a class can be added, deleted, and replaced. Figure 1 is an example of an OC.
- RN: This 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.
- RL: This is a JPM for changing the relation between two classes, in which case, a join point is a class, and advice adds an aggregation and a relationship between two classes selected by a pointcut.
- IH: This is a JPM for changing the inheritance between two classes, in which case, a join point is a class, and advice adds an inheritance between two classes selected by a pointcut.

Although Core AspectM only supports class diagrams, the concept can be applied to other diagrams such as statecharts and sequence diagrams. Moreover, the dynamic aspect of system behavior can be described as a protocol state machine, because a modeler can specify preconditions and postconditions in an operation by using OCL (Object Constraint Language) [39].

2.1.2 Language Features

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

Figure 2 shows an example of the Core AspectM diagram notations and the corresponding XML formats. Core AspectM provides the two kinds of aspects: an ordinary aspect and a component aspect. A component aspect is a special aspect used for com-
posing aspects. 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 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, three kinds of predicates can be used: $\text{cname}$ (class name matching), $\text{aname}$ (attribute name matching), and $\text{oname}$ (operation name matching). Three logical operations can be also used: && (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. Advice is applied at join points selected by a pointcut. The precedence of aspect application can be specified using a dependency association. The precedence is important, because the result of weaving may be ambiguous when multiple aspects can be applied to the same join point.

2.2 Problems in Core AspectM

Although Core AspectM provides basic JPMs, a modeler cannot define domain-specific JPMs. It would be better for a modeler to describe a model as shown in Fig. 3 (the class diagram is cited from [10]). Domain-specific model elements are denoted by stereotypes that are not merely annotations but elements introduced by metamodel extension.

The model in Fig. 3 describes an invoice processing system comprised of two kinds of domain-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 a domain-specific JPM DCLogger that adds log operations to DCEntityContracts whose UniqueId is not assigned by users. DCLogger consists of domain-specific pointcut designators and advice. DCEntityContract can be regarded as a domain-specific join point. The DCEntityContract.UniqueId.isUserAssigned pointcut selects two classes Customer and Invoice. If only primitive predicates provided in Core AspectM can be used, it is necessary to specify as follows: $\text{cname}($"Customer") || $\text{cname}($"Invoice"). In this case, a modeler has to pick up all class definitions and check whether they include an...
isUserAssigned tag and its value is false. This activity is prone to error. The definition cname("Customer") || cname("Invoice") must be modified whenever an isUserAssigned tag value is changed. Moreover, we have to change this pointcut definition when we add a new class having an isUserAssigned=false tag. As mentioned here, pointcut definitions based on Core AspectM tend to be fragile in terms of software evolution. On the other hand, an expressive pointcut such as !DCEntityContract.UniqueId.isUserAssigned is robust, because this pointcut definition does not have to be modified even if the isUserAssigned tag value is changed.

Although expressive and domain-specific JPMs are effective, 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, the following fact cannot be specified: DCEntityContract must have a UniqueId whose tag is isUserAssigned. If an aspect is defined 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 domain-specific models precisely.

There are situations that need domain-specific JPMs—for example, domain-specific logging, domain-specific resource management, domain-specific transaction, and so on.

3. Metamodel Access Protocol

To deal with the problems in Sect. 2, MMAP has been introduced. In this section, we illustrate the outline of MMAP [34]. Current AspectM includes not only language constructs provided by Core AspectM but also MMAP that can extend the AspectM metamodel as shown in Fig. 4.

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 profiles provide stereotypes for introducing rich vocabularies. While a modeler can easily introduce some kinds of domain-specific JPMs—for example, pointcut definitions using stereotypes such as UniqueId—there are situations in which stereotypes as mere annotations are insufficient: the typing of tags is weak; and new associations among UML metamodel elements cannot be declared [10]. For example, as explained in 2.2, stereotypes

![Fig. 4 Part of AspectM metamodel.](image-url)
as annotations cannot describe relations among domain-specific model elements including DCEntityContract, DCControllerContract, UniqueId, and DCLogger: the relations among them are described in the metamodel.

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 ordinary modeler to extend the UML metamodel by using the full power of the MOF.

MMAP is a middle weight approach that restricts available extension by MOF. MMAP aims at introducing a domain-specific JPM that cannot be described by UML stereotypes. A modeler can access and modify the AspectM metamodel by using protocols exposed by MMAP. The target of MMAP is not a tool developer that needs full access to the AspectM metamodel but an ordinary 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.

### 3.2 Metamodel

Figure 4 shows a part of the AspectM metamodel defined as an extension of the UML metamodel. The aspect (AspectComponent) class inherits the Classifier class. Pointcuts and advice are represented by the Pointcut class and the Advice class, respectively. Concrete advice corresponding to the seven JPMs is defined as a subclass of Advice: PointcutAndAdvice for PA, Composition for CM, NewElement for NE, OpenClass for OC, Rename for RN, Relation for RL, and Inheritance for IH. The Pointcut class is common to all JPMs, because pointcuts can be specified uniformly. The constraints among metamodel elements can be specified in OCL.

### 3.3 Protocols

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 include Class, Attribute, Operation, Association†, and a set of JPM metaclasses. In Fig.4, a class represented by a gray box is an extension point. Class, Attribute, Operation, and Association are necessary to extend the UML class definitions. PointcutAndAdvice, Composition, NewElement, OpenClass, Rename, Relation, and Inheritance are necessary to extend AspectM advice mechanisms. Currently, Pointcut is not exposed as an extension point, because user-defined pointcuts can be specified using primitive predicates shown in Table 2. An extension operation is a modeling activity allowed at the exposed extension points. There are four operations including define subclasses, add attributes to subclasses, create associations among subclasses, and add/delete/replace constraints. Table 2 is a list of primitive predicates for navigating the metamodel. Using these predicates, pointcut designators can be defined as below. The defined pointcut designator represents all elements that satisfy the right-hand side predicates.

<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>
<tr>
<td>precondition-of(x, o)</td>
<td>x is precondition of an operation o</td>
</tr>
<tr>
<td>postcondition-of(x, o)</td>
<td>x is postcondition of an operation o</td>
</tr>
<tr>
<td>invariant-of(x, c)</td>
<td>x is invariant of a class c</td>
</tr>
</tbody>
</table>

A modeler can add constraints consisting of preconditions, postconditions, and invariants to his or her model. Preconditions and postconditions and invariants can be specified to an operation and a class, respectively. The constraints specified in OCL can be also captured by defining pointcut designators using primitive predicates.

The design of MMAP is similar to that of application frameworks in which hot-spots should be exposed. By using MMAP, a modeler does not have to 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 metaclass protocol (MOP) [19] and computational reflection [22], [29] 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.

### 3.4 Challenges in MMAP Implementation

Although MMAP can improve the expressiveness of AOM, MMAP needs a flexible development environment that supports reflection in modeling. However, it is not easy to support modeling-level reflection because of the following reasons: 1) it is not clear how to implement modeling-level reflection; and 2) MMAP affects all aspects of tool implementation. Reflection is implemented in some programming languages in which language extension is specified as program code itself. In the same way, we think that language extension in modeling should be specified in the modeling activities. That is, a model editor should support reflection. Moreover, it is desirable to specify some kinds of meta-level constraints to support correct modeling using a model editor. In most reflective programming languages, language

†Association is omitted in Fig. 4.
extension has to be dealt with at the execution time. However, it is desirable that language extension is dealt with at the weaving time (compile time), because a model has to be translated into not only reflective programming languages but also ordinary programming languages. To support this kind of weaving, a model weaver should be extensible, because it should be able to deal with user-specified language extension at the weaving time.

We have to deal with following challenges in order to implement MMAP effectively.

- A model editor should be able to edit new model elements introduced by extending the metamodel. These model elements are not merely graphic components but are constrained by OCL descriptions specified in the metamodel.
- A model weaver should be able to capture new models elements as join points and deal with new pointcuts defined by MMAP. The weaver should be extensible.
- The correctness of model weaving should be verified, because it is difficult to check the consistency and the aspect interference due to the metamodel extension.

To solve the issues, we introduce the notion of edit-time structural reflection (or model-time structural reflection), extensible model weaving, and weaving verifier.

4. AspectM Support Tool

We have developed a proof-of-concept implementation of the AspectM support tool consisting of model editor, model weaver, and model verifier.

Figure 5 illustrates the tool overview. A developer creates UML and aspect models using the model editor that supports edit-time structural reflection. We call this editor reflective model editor. Using the model editor, a developer can use domain-specific notations introduced by the reflective mechanism. After that, a set of class and aspect models are woven into an ordinary class model by the model weaver. This model weaver can generate Java program from a woven class model. This weaver is extensible and can deal with aspects introduced by the reflective model editor. The model verifier checks the followings: 1) UML and aspect models should conform to the metamodel; and 2) the result of weaving should reflect the intention of a modeler. The functions of the model verifier are embedded in the model weaver. A modeler does not have to be aware of the existence of the model verifier.

We explain the details of each tool in Sect. 5, 7, and 8. The essence of reflective model editor, extensible model weaver, and model verifier is not limited to AspectM but can be applied to other kind of modeling languages such as ordinary UML. Although the implementation of language features specific to aspect orientation such as pointcut & advice depends on AOM, the implementation method is not limited to AspectM but can be applied to other kinds of AOM languages.

5. Reflective Model Editor

In this section, the reflective model editor and the metamodel extension procedures using this editor are shown.

5.1 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. Figure 6 is a screen shot that edits the invoice processing system (left side) in...
There are two kinds of modelers that use the reflective mechanism. One is an application modeler. Another is a modeler that develops DSLs. In the former case, an application modeler not only creates an application model but also extends the meta model to describe application-specific modeling notations or constraints. The total of application model consists of application descriptions and their meta descriptions. This idea is similar to the notion of language-oriented programming (LOP) [38] in which a programmer defines DSLs for the problem first, and solves the problem using these DSLs. In such a case, a meta-level description can be regarded as one of the modules consisting of applications. We think that our approach can be called language-oriented modeling (LOM). This kind of meta-level descriptions is strongly related to an applications and are introduced in order to improve the application expressiveness. In the latter case, language features that can be commonly reused are defined by a DSL developer. An ordinary application modeler creates a model only using such a pre-defined DSL. In this case, the application modeler does not have to be aware of MMAP.

5.1.1 Edit-Time Structural Reflection

As illustrated in Fig. 7, the edit-time structural reflection consists 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 MMAP primitive predicates. The metamodel editor exposes extension points. Only extension points are displayed on the editor screen as shown in Fig. 6. 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 reflection. 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 reflection. In reflective programming, a programmer can introduce new language features using MOP. In our approach, a modeler can introduce new model elements using MMAP.
5.1.2 Metamodel Extension Procedure

Using the example of the invoice processing system, we illustrate a procedure for extending the AspectM metamodel. As mentioned in 2.2, the Logging aspect in Fig. 6 (left side) adds a log operation to the DCEntityContracts components whose UniqueId is not assigned by users. Although the bodies of the logClasses pointcut and the addLog advice whose type is OC<<DCLogger>> are invisible in Fig. 6, they are defined as same as Fig. 3.

The outline of extension steps is as follows: 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.

5.1.3 Domain-Specific Model Element

Extension operations are executed at exposed extension points in order to introduce new domain-specific model elements as shown in Fig. 8. The constraints among new model elements can be specified using OCL. The model elements that violate the OCL descriptions can be detected by the editor. For example, the following constraint is defined in UniqueID meta class.

\[ \text{self.owner.feature} \rightarrow \text{select(ocIsTypeOf(UniqueId))} \rightarrow \text{size()} \leq 1 \]

This constraint specifies that a class can have at most one unique identifier. If a modeler creates a class having two identifiers, the model editor displays an error message as shown in the bottom of Fig. 9.

5.1.4 Domain-Specific Pointcut Designator

Pointcut designators are defined as below.

\[ \text{define pointcut} \]
\[ \text{DCEntityContract}\_\text{UniqueId\_isUserAssigned}(c): \]
\[ \text{meta-class-of} ("DCEntityContract", c) \land \]
\[ \text{member-of}(a, c) \land \]
\[ \text{meta-class-of} ("UniqueId", a) \land \]
\[ \text{member-of} ("isUserAssigned", "UniqueId") \land \]
\[ \text{value-of} ("true", "isUserAssigned") \]

This pointcut designator selects all classes that match the following conditions: 1) the metaclass is DCEntityContract; 2) the value of the isUserAssigned is true. In case of Fig. 3, the negation of this pointcut designator selects the two classes Customer and Invoice. Figure 10 shows an image of metamodel navigation using MMAP predicates above.

5.1.5 Plug-In Components for Domain-Specific Modeling

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.

5.2 Implementation

The reflective model editor, a plug-in module for Eclipse, is developed using the Eclipse Modeling Framework (EMF) [9] and Graphical Modeling Framework (GMF) [11].
Fig. 10  Metamodel navigation using MMAP predicates.

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.

6. Model Weaver

In this section, we show how to construct the extensible model weaver.

6.1 Concept

The model weaver compounds a set of base models consisting of classes and aspects. This weaver has to be able to deal with a variety of DSLs defined by using edit-time reflection. Pointcut analysis plays an important role in weaving. In edit-time reflection, a domain-specific pointcut designator is defined by using MMAP predicates. In the AspectM weaver, application models, meta models, and pointcut designators are translated into a set of predicates defined by Prolog facts in order to implement pointcut analysis easily.
6.2 Implementation

The model weaver is implemented using DOM (Document Object Model) and Prolog. After model weaving, a model is translated into a source program. Currently, Java code can be generated. As shown in 2.1.2, an AspectM model is represented in the form of XML. DOM is used to handle XML-based AspectM models and transform them to a set of Prolog facts.

First, the model weaver transforms the base and meta models to a set of Prolog facts. For example, the Invoice class and related metamodel elements are represented as follows.

```xml
-- from Invoice class
meta-class-of("DC EntityContract", "Invoice"),
member-of("number", "Invoice"),
meta-class-of("UniqueId", "number"),
value-of("true", "isUserAssigned").
```

```xml
-- from AspectM metamodel
member-of("isUserAssigned", "UniqueId").
```

Second, the model weaver converts a pointcut into a Prolog query, and checks whether the query satisfies the facts above. For example, the negation of the DC EntityContractUniqueId.isUserAssigned pointcut selects Customer and Invoice as join points. The model weaver executes advice at these join points. JPL, a set of Java classes providing an interface between Java and Prolog, provided by SWI-Prolog [31] is used for bridging the model weaver and the Prolog interpreter.

In current MMAP, the Advice class is not exposed as an extension point, because this extension needs a new weaver module that can handle a new kind of advice. Adopting our approach, the model weaver does not have to be modified even if the metamodel is modified by the reflective model editor.

As shown in this section, the model weaver can deal with domain-specific join points and pointcuts introduced by using MMAP. That is, our model weaver is extensible.

7. Model Verifier

In this section, we show a verification mechanism for the extensible model weaving.

7.1 Concept

It is important to clarify what kind of model weaving should be verified. We focus on the followings: 1) UML and aspect models should conform to the metamodel structurally; and 2) the result of weaving should reflect the intention of a modeler.

7.1.1 Structural Correctness

There are three kinds of problems on the structural correctness. First, a base model might not conform to the modified metamodel even if the base model conforms to the previous metamodel. For example, a base model that includes a class instantiated by a metaclass introduced by a metamodel (version 1) does not conform to a new metamodel (version 2) if the metaclass is deleted in the version 2. Second, a woven model might not conform to the metamodel even if each base model before weaving conforms to the metamodel. Since a new kind of model transformation can be introduced by adding user-defined aspects, a model transformed by these aspects might not conform to the metamodel if the aspect descriptions are not adequate. Third, the interference among aspects is an important problem although it is not inherent to extensible modeling languages. A woven model might include unexpected bugs due to the aspect inference such as name conflicts, multiple inheritance, and cyclic inheritance.

7.1.2 Intention of a Modeler

Although the mechanism of user definable pointcut designators is effective, it is not easy for a modeler to check whether an introduced pointcut designator captures join points correctly. Captured join points might not be the points intended by the modeler. Since a pointcut designator is defined using primitive predicates based on first order logic, the intention of the modeler can be specified as a set of assertions based on first order logic.

The precedence of aspect application is also an important problem. Although the precedence can be specified in AspectM, the intended results might not be obtained when a modeler makes a mistake in specifying the precedence. The mixture of wrong pointcut definitions and wrong aspect precedence specifications might cause an unexpected weaving. A pointcut may capture wrong join point even if its definition seems correct. It is difficult to find a defect without a verification mechanism.

7.2 Implementation

The model verifier consists of the metamodel checker for verifying whether a base model conforms to the metamodel, the model structure checker for detecting the aspect interference, and the assertion checker.

7.2.1 Metamodel Checker

The metamodel checker is based on XML Schema, because base and meta models are stored as XML documents. Using XML Schema, we can check whether a base model conforms to the metamodel.

The verification procedure is as follows: 1) generates the default schema definition from the original metamodel that is not yet extended; 2) checks whether the metamodel is extended; 3) redefines the schema definition if extended; and 4) checks whether a base model conforms to the extended metamodel.
7.2.2 Model Structure Checker

The model structure checker is implemented using DOM. The checker parses a model as an XML document, and detects the name conflicts (classes, attributes and operations that have the same names), multiple inheritance, and cyclic inheritance.

7.2.3 Assertion Checker

In the assertion checker, an assertion is specified using the primitive predicates as shown in Table 3.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>super_class_of(c1,c2)</td>
<td>c1 is a superclass of c2</td>
</tr>
<tr>
<td>attribute_of(a,c)</td>
<td>a is an attribute of c</td>
</tr>
<tr>
<td>operation_of(o,c)</td>
<td>o is an operation of c</td>
</tr>
<tr>
<td>advice_of(a,o)</td>
<td>a is woven into o</td>
</tr>
<tr>
<td>class_exist(c)</td>
<td>c exists in a model</td>
</tr>
<tr>
<td>related_to(c1,c2)</td>
<td>c1 is related to c2</td>
</tr>
<tr>
<td>aggregate(c1,c2)</td>
<td>c1 aggregates c2</td>
</tr>
<tr>
<td>composed_of(c1,c2)</td>
<td>c1 is composed of c2</td>
</tr>
</tbody>
</table>

The verification procedure is as follows: 1) translates base and meta models into Prolog facts; 2) generates Prolog queries from assertions; and 3) checks the satisfiability of the Prolog queries. Since an AspectM model is stored as an XML document, the step 1 can be implemented as a translator from XML to Prolog. The following is an example of an XML model and generated Prolog facts. This model represents an operation TransOp whose type is TransactionOperation.

```xml
<ownedElement name="TransOp"
xsi:type="asm:TransactionOperation" />
```

-- Generated Prolog facts

```prolog
modelElement(
    [property('tagName', 'ownedElement'),
     property('name', 'TransOp'),
     property('xsi:type', 'asm:TransactionOperation')])
```

When a modeler intends to add the TransOp operation to the C class by applying an OC (open class) aspect and wants to check whether the aspect really performs the job, the modeler has to specify the following assertion.

`operation_of('TransOp','C')`

If this assertion is true, the join point capturing and the advice weaving satisfy the intention of the modeler. Of course, this is trivial when a model includes this OC aspect only. However, in most case, there are multiple aspects in a model. An aspect may interfere with this OC aspect and the result of weaving may not include the TransOp operation.

In the model verifier, the default assertions that should be satisfied after weaving can be automatically generated from aspect definitions, because the assertions are given as the patterns as shown in Table 4. A modeler can check whether a weaver handles aspects correctly even if he or she does not give any assertions. The assertion above can be automatically generated. The generated assertion is useful for unit testing of an aspect, because the effect of the aspect application might be different from the intention of a modeler. That is, the aspect might not be properly dealt with due to aspect interference. When the result of a weaving does not satisfy the intention of a modeler, the modeler can examine which aspect is inadequate using generated assertions.

Although the default assertion generator is effective, all of the assertions that represent the intention of a modeler cannot be generated. The intention concerning the semantics of a model must be specified by a modeler.

8. Case Study

In this section, we show a case study using the AspectM support tool. We show how to construct a DSL support tool called CAMEmb tool. CAMEmb (Context Analysis Method for Embedded systems) [36], [37] is a requirements analysis method for constructing reliable embedded systems. First, we explain CAMEmb briefly. After that, we show how to construct CAMEmb model editor and weaver using MMAP. We use a LEGO line trace car as an application example of the CAMEmb tool. The car runs tracing a line by observing a line color. Lastly, we evaluate the effectiveness of our approach.

8.1 CAMEmb Overview

Currently, development of embedded systems is mainly conducted from the viewpoint of system functionalities: how hardware and software components are configured to construct a system—contexts are not considered explicitly in most cases. As a result, unexpected behavior might emerge in a system if a developer does not recognize any possible external contexts. It is important to analyze external contexts in order to detect the unfavorable behavior systematically at the early stage of the development.

CAMEmb is a requirements analysis method for dealing with the above problems. In CAMEmb, a context model

---
†These predicates are slightly different from the MMAP primitive predicates for describing assertions, we provide predicates specific to assertion descriptions for convenience.
is constructed using a DSL based on UML Profile for Context Analysis shown in Table 5. This profile can describe system elements, context elements, and associations between them: three kinds of stereotypes ≪ Context ≫, ≪ Sensor ≫, and ≪ Actuator ≫ are defined as an extension of the UML class; and five kinds of stereotypes ≪ Observe ≫, ≪ Control ≫, ≪ Transfer ≫, ≪ Noise ≫, and ≪ Affect ≫ are defined as an extension of the UML association. The arrow of ≪ Observe ≫ and ≪ Control ≫ indicates the target of observation and control. The arrow of ≪ Noise ≫ and ≪ Affect ≫ indicates the source of noise and affect, respectively. The arrow of ≪ Transfer ≫ indicates the source of transformation.

### Table 5: Model elements for context analysis.

<table>
<thead>
<tr>
<th>Model element</th>
<th>MMAP extension point</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>≪ Context ≫</td>
<td>Class</td>
<td>Context element</td>
</tr>
<tr>
<td>≪ Hardware ≫</td>
<td>Class</td>
<td>Hardware element</td>
</tr>
<tr>
<td>≪ Software ≫</td>
<td>Class</td>
<td>Software element</td>
</tr>
<tr>
<td>≪ Observe ≫</td>
<td>Association</td>
<td>Hardware observes a context</td>
</tr>
<tr>
<td>≪ Control ≫</td>
<td>Association</td>
<td>Hardware controls a context</td>
</tr>
<tr>
<td>≪ Noise ≫</td>
<td>Association</td>
<td>Noise from a context</td>
</tr>
<tr>
<td>≪ Transfer ≫</td>
<td>Association</td>
<td>Data is transformed into different form</td>
</tr>
<tr>
<td>≪ Affect ≫</td>
<td>Association</td>
<td>Data from the target context is affected by other context</td>
</tr>
</tbody>
</table>

Our UML profile focuses on sensing and actuating.

### 8.2 Model Editor Construction

Figure 11 is a snapshot of CAMEmb model editor. The notations supporting UML Profile for Context Analysis are introduced. A developer can create a model using this profile. New model elements supporting CAMEmb are introduced by using MMAP as shown in the top of Fig. 11. The bottom of Fig. 11 illustrates the result of the context analysis for a line trace car observing the light reflected from the ground. Constraints among metamodel elements are specified in OCLs as follow.

```ocl
context ≪ Actuator ≫
self.ownedAssociation->select(oclIsTypeOf(Control))
```

This OCL indicates that ≪ Actuator ≫ should have ≪ Control ≫, and can be verified by our extended model editor. OCLs can be also specified in the base-level and captured by defining pointcut designators using MMAP. For example, we can capture the contexts whose illuminance is larger than 1000 as below. This is an example of domain-specific pointcuts.

```ocl
define pointcut
Illuminance_is_greater_than_1000(c):
```

![Fig. 11](image-url) An extended model editor that supports a context analysis DSL.
Fig. 12 Generated software design model.

Table 6 Aspects for model transformation.

<table>
<thead>
<tr>
<th>JPM (Advice)</th>
<th>Transformation</th>
<th>NOA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. RL (modify-relationship)</td>
<td>Change navigation directions</td>
<td>1</td>
</tr>
<tr>
<td>2. RN (replace-name)</td>
<td>Rename sensors and actuators to transform them into driver classes</td>
<td>1</td>
</tr>
<tr>
<td>3. OC (add-operation)</td>
<td>Add operations to driver classes</td>
<td>3</td>
</tr>
<tr>
<td>4. OC (add-operation)</td>
<td>Add operations to context classes</td>
<td>8</td>
</tr>
<tr>
<td>5. EL (modify-class)</td>
<td>Remove stereotypes because they are meaningless in a design model</td>
<td>1</td>
</tr>
</tbody>
</table>

* NOA (number of aspects)

```
meta-class-of('Context', c) &&
invariant-of('external_illuminance>1000', c)
```

8.3 Model Weaver Construction

CAMEmb model weaver transforms context analysis models into a software design model that takes into account the contexts as shown in Fig. 12. The Driver and Context Recognition layers are automatically generated by the weaver. Although the Controller layer is created by hand currently, this can be transformed from a system analysis model. A set of contexts in the context analysis models is transformed into internal state classes recognized by a software controller. The Driver and Actuator layers are transformed into driver classes that operate hardware components. The software controller interacts with expected contexts by referring and changing the values of the internal state classes that communicate with driver classes.

CAMEmb model weaver can be constructed by defining a set of aspects shown in Table 6. These aspects crosscut over multiple model elements. For example, the aspect with the replace-name advice (No.2) captures sensors/actuators (LightSensor, DriveMotor, and SteerMotor) as join points and transforms them into driver classes. The definition of this pointcut is as follows: `meta-class-off('Actuator',c) || meta-class-off('Sensor',c)`.

Using model verifiers, we can check the followings: 1) the context analysis model conforms to the metamodel extended for supporting CAMEmb; and 2) the generated software design model is well-formed. The intention of a modeler can be also checked. For example, if the modeler wants to check whether the software controller can access to driver classes, he or she only has to specify an assertion such as “related-to('ControlSW',d) && meta-class-of('Driver',d)”. If this assertion is violated, an error message is displayed.

To develop a reliable line trace car, we have to reduce the effect of EnvironmentLight. For example, we can define an aspect below that captures the contexts whose illuminance is larger than 1000 and replaces their invariants to “inv : external_illuminance < 1000”. This aspect corresponds to a system design that reduces the effect of the illumination by adding a shade. Since there are multiple aspects for dealing with non-functional requirements such as reliability, we must take into account the interference among aspects. Our verifier is effective for this purpose.

```
strongIllumination : context{
  pointcut-body :=
  Illuminance_is_greater_than_1000(*)
}
shadeLight[strongIllumination]:
  replace-invariant{
    advice-body :=
    inv: external_illuminance < 1000
  }
```

8.4 Evaluation

In 3.4, we pointed out three research challenges in MMAP implementation: 1) the model editor should be able to edit new model elements introduced by extending the metamodel; 2) the model weaver should be able to capture new models elements as join points and deal with new pointcuts defined by MMAP; and 3) the correctness of model weaving should be verified.

As demonstrated in this section, CAMEmb tool can be constructed using the reflective model editor and the extensible model weaver. The model verifier can check the correctness of a context model. 1), 2), and 3) are systematically implemented.

Table 7 shows the number of tasks for defining DSL and constructing the CAMEmb weaver. Three aspects (No.1, 2, 5 in Table 6) are not specific to the line trace car

<table>
<thead>
<tr>
<th>Task Description</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Extend the metamodel</td>
<td>8</td>
</tr>
<tr>
<td>2. Define meta-level OCLs</td>
<td>8</td>
</tr>
<tr>
<td>3. Define aspects commonly reused in CAMEmb</td>
<td>3</td>
</tr>
<tr>
<td>4. Define aspects specific to a line trace car</td>
<td>11</td>
</tr>
<tr>
<td>5. Reuse LEGO OS components such as lejos.nxt.LightSensor</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 7 Tasks for constructing DSL tool.
development. Although other eleven aspects are specific to the line trace car, they can be reused repeatedly if software product line (SPL) [18] approach is adopted. In SPL, a product is constructed by assembling core assets, components reused in a family of products (line trace car family in this case).

The code size of the line tracing function was 223 LOC (lines of code), and 174 LOC was automatically generated from the context analysis model by using the CAMEmb model weaver. The percentage of automated generated code was 78%. Since the classes in the context recognition layer are value object classes, these classes only need setters, getters, and calls to other connected context classes. It is easy to generate these programs. Since the driver classes only call the LEGO OS components, it is also easy to generate code. However, only the software controller program had to be coded by hand. This can be generated from system analysis model if our weaver supports state machine diagrams. We plan to develop a model weaver for state machine diagrams.

The CAMEmb tool can support model-driven architecture (MDA) [23] in which platform-independent models (PIM) are transformed to platform-specific models (PSM). In this case study, the platform was LEGO OS and the model transformation could be constructed by only using MMAP. No extra programming was needed. We did not have to modify the metaclasses that were not exposed by MMAP. From our experience, MMAP was sufficient to extend the AspectM metamodel. We believe that the MOP approach in modeling-level is more effective than the full metamodel extension approaches in terms of the cost and usability.

9. Related Work

In this section, we discuss modeling-level aspect orientation, DSL, UML2 profiling, edit-time reflection, logic-based pointcut definitions, verification of weaving, and model transformation languages in reference to related work.

9.1 Aspects in Modeling

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 [30]. 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 [16]. An aspect in AspectM is not mapped to an element of a specific programming language, but operates on UML diagrams. J. Sillito et. al. proposed the concept of usecase-level pointcuts, and showed the effectiveness of JPMs in early modeling phases [28]. We think that the notion of JPMs in AspectM can be applied to usecase diagrams. In Motorola WEAVR [5], weaving for UML statecharts including action semantics can be possible. In AspectM, behavioral models can be described as protocol state

9.2 Domain-Specific AO Language

As demonstrated in this paper, 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 [20]. M. Shonle et. al. proposed an extensible domain-specific AOP language called XAspect that adopted plug-in mechanisms [27]. 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.

Domain-specific extensions are necessary not only at the programming stage but also at the modeling stage. J. Gray proposed a technique of aspect-oriented domain modeling (AODM) [12], [13] that adopted the Generic Modeling Environment (GME), a meta-configurable modeling framework. He also proposed an approach that used a program transformation system as the underlying engine for weaver construction [14]. The GME provides metamodeling 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 midlewight. Although all of the AspectM metamodel cannot be extended, domain-specific model elements can be introduced at relatively low cost.

9.3 UML 2 Profiling

Table 8 compares MMAP with UML profile. In UML2, profiling mechanisms that have been improved from UML 1.x include part of our ideas, because new kinds of stereotypes can be introduced by extending UML metamodel and OCL can be specified. However, unfortunately, most of the current UML2 tools still provide only the stereotype definition support. Moreover, these tools do not take into account the extensibility from the viewpoint of model compilation. The notion of structural reflection does not exist explicitly in UML2 profiling. Main contribution of our approach is to provide an extensive mechanism for AOM. This includes extension of model elements, join points, pointcuts, and verification. We also provide a concrete implementation strategy for realizing an integrated extensible AOM environment. Of course, our tool can be used as a substitute of the

<table>
<thead>
<tr>
<th>Table 8</th>
<th>MMAP vs. UML profile.</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.</td>
<td>MMAP</td>
</tr>
<tr>
<td>1.</td>
<td>Extension mechanism</td>
</tr>
<tr>
<td>2.</td>
<td>OCL support</td>
</tr>
<tr>
<td>3.</td>
<td>Metamodel navigation</td>
</tr>
<tr>
<td>4.</td>
<td>Aspect support</td>
</tr>
<tr>
<td>5.</td>
<td>Verification</td>
</tr>
</tbody>
</table>
UML profiling facility. If we take a stand on UML profiling, we can consider our approach a promising mechanism for integrating AOM with UML profiles.

In our current implementation, we do not use OCL but Prolog to provide predicates for metamodel navigation. This is because it is relatively easy to implement the reasoning and verification mechanisms in our prototype tool. However, we plan to change Prolog to OCL in the near future.

9.4 XML and AO Language

AspectM can be considered an XML-based aspect-oriented 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 [40]. M. Eichberg, M. Mezini, and K. Ostermann investigated the use of XQuery for specification of pointcuts [6].

9.5 Edit-Time Reflection

MMAP is similar to an edit-time metasocket protocol (ETMOP) [7, 8] proposed by A.D. Eisenberg and G. Kiczales. ETMOP runs as part of a code editor and enables meta data 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 is to provide mechanisms for making programs more visually expressive is similar to our goal, we focus on the provision of middleware mechanisms for domain-specific expressiveness.

9.6 Logic-Based Pointcut Definition

K. Gybels and J. Brichau proposed pattern-based pointcut constructs using logic programming facilities [15]. K. Ostermann, et al. also proposed a pointcut mechanism based on logic queries written in Prolog [24]. 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.

9.7 Verification of Weaving

The verification of a weaving is one of the important research topics in AOP. For example, W. Havinga, et al. proposed a graph-based approach to modeling and detecting composition conflicts related to introductions [17]. They provided a prototype tool that detects and visualizes the occurrence of conflicts in AspectJ programs. S. Shinotsuka, et al. proposed the concept of WbC (Weaving by Contract) [26], a technique to verify weaving based on contracts. Contracts in WbC consist of preconditions, postconditions, and invariants. A precondition states under which conditions weaving can be applied. A postcondition states what condition is verified after weaving has been accomplished, and an invariant states what conditions weaving preserves. Introducing WbC, we can specify how a program should behave before and after weaving.

Although the verification mechanisms targeted to AOP have been explored, the modeling-level verification are not yet discussed enough. In this paper, we clarified how the metamodelling affects the verification of the weaving.

9.8 Model Transformation Language

Currently, standard model transformation languages such as QVT (Queries, Views, and Transformations) [25] and ATL (ATLAS Transformation Language) [3] are proposed. ATL is developed to answer the QVT Request For Proposal. In QVT, model elements to be transformed are selected by query facilities based on OCL, and are converted using transformation descriptions. QVT is powerful, and promotes the componentization of model compilers. Although QVT is powerful, it is not easy for a modeler to write transformation rules. The main target user of QVT is not a modeler but a developer of model compilers. However, there are many situations that a modeler wants to define application-specific transformation rules such as optimizations. For example, there is a case in which classes consisting of a system should be reorganized in terms of memory resource limitations: unused classes or methods are deleted; and classes assigned to the same node are merged into a single class.

It would be easy for a modeler to define model transformation rules if the modeler can specify the rules at the same level of ordinary UML modeling. Adopting our approach, we can practice a language-oriented modeling introduced in 5.1†.

10. Conclusion

In this paper, we show how to implement an extensible AOM environment. The notion of MMAP plays an important role in constructing domain-specific AOM languages. In order to support MMAP, we introduced the notion of edit-
time structural reflection, extensible model weaving, and verification mechanisms. In our current implementation, we do not use OCL but Prolog, because it is relatively easy to implement the reasoning and verification mechanisms in our prototype tool. Currently, we are developing a new version of AspectM in which OCL is adopted. Although our current tool needs refining, it is an important step towards extensible AOM environments integrated with UML.

Acknowledgements

We thank Yusaku Maeno, Satoshi Murakami, Shinji Sano, Yuuki Ito, Jun Yoshida, Kazuhide Noda, Genya Otsubo, Daisuke Matsushita, Hiroshi Kanagawa, Toshiki Seto, Susumu Taniguchi, and Tomoya Koga for their great contributions. They were students of Naoyasu Uabayashi when he worked for Kyushu Institute of Technology. Yusaku Maeno, Satoshi Murakami, Shinji Sano, and Yuuki Ito developed a preliminary version of AspectM support tool. Shinji Sano developed the model editor. Yusaku Maeno developed the model weaver. Genya Otsubo, Kazuhide Noda, and Daisuke Matsushita enhanced the preliminary version and implemented MMAP. Genya Otsubo developed the reflective model editor. Kazuhide Noda implemented verification mechanisms. Jun Yoshida constructed CAMEmb tool using AspectM support tool. Hiroshi Kanagawa, Toshiki Seto, Susumu Taniguchi, and Tomoya Koga developed CAMEmb. All of the students graduated from Kyushu Institute of Technology a few years ago. Currently, they are working as software engineers.

This research is being conducted as a part of the Grant-in-aid for Scientific Research (B), 23300010 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References


[40] XQuery, http://www.w3.org/TR/xquery/

Naoyasu Ubayashi is a professor at Kyushu University since 2010. He is leading the POSL (Principles of Software Languages) research group at Kyushu University. Before joining Kyushu University, he worked for Toshiba Corporation and Kyushu Institute of Technology. He received his Ph.D. from the University of Tokyo. He is a member of ACM SIGPLAN, IEEE Computer Society, and Information Processing Society of Japan (IPSJ). He received “IPSJ SIG Research Award 2003” from IPSJ.

Yasutaka Kamei received the B.E. degree (2005) in Informatics from Kansai University, Japan, and the M.E. degree (2007) and Ph.D. degree (2009) in Information Science from Nara Institute of Science and Technology, Japan. He has been a research fellow of the Japan Society for the Promotion of Science (PD) from July 2009 to March 2010. From April 2010 to March 2011, he was a postdoctoral fellow at Queen’s University in Canada. He is currently an assistant professor in Graduate School and Faculty of Information Science and Electrical Engineering at Kyushu University in Japan. His research interests include empirical software engineering, open source software engineering and mining software repositories (MSR).