Leveraging Metamodeling and Attribute-Oriented Programming to Build a Model-driven Framework for Domain Specific Languages[*]

Hiroshi Wada
hiroshi_wada@otij.org
Object Technology Institute, Inc.
Akasaka, Minato-ku,
Tokyo, 107-0052, Japan

Junichi Suzuki
jxs@cs.umb.edu
Department of Computer Science
University of Massachusetts, Boston
Boston, MA 02125-3393, USA

Shingo Takada
michigan@doi.ics.keio.ac.jp
Graduate School of Science and Technology
Keio University
Yokohama City, 223-8522, Japan

Norihisa Doi
doi@ise.chuo-u.ac.jp
Faculty of Science and Engineering,
Chuo University
Tokyo, 112-8551, Japan

ABSTRACT
This paper proposes a new model-driven framework that allows developers to model and program domain-specific concepts (ideas and mechanisms specific to a particular domain) and to transform them toward the final (compilable) source code in a seamless manner. The proposed framework provides an abstraction to represent domain-specific concepts at both modeling and programming layers by leveraging the notions of UML metamodeling and attribute-oriented programming. This paper describes the design and implementation of the proposed framework, and discusses how the framework can improve the productivity to implement domain-specific concepts and how it can increase the longevity of models and programs representing domain-specific concepts. In order to demonstrate how to exploit the proposed framework, this paper also shows a development process using an example DSL to specify service-oriented distributed systems.

1. INTRODUCTION

Software modeling is becoming a critical process in software development. Modeling technologies have matured to the point where it can offer significant leverage in all aspects of software development [1]. For example, the Unified Modeling Language (UML) provides a rich set of modeling notations and semantics, and allows developers to understand, specify and communicate their application designs at a higher level of abstraction [2]. The notion of model-driven development aims to build application design models and transform them into running applications [3]. Given modern modeling technologies, the focus of software development has been shifting away from technology domains toward the concepts and semantics of problem domains. The more directly application models can represent domain-specific concepts, the easier it becomes to specify applications. One of the goals of modeling technologies is to map modeling concepts directly to domain concepts [4].

Domain Specific Language (DSL) is a promising solution to directly capture, represent and implement domain concepts [5, 6]. DSLs are languages targeted to particular problem domains, rather than general-purpose languages that are aimed at any software problems. Each DSL provides built-in abstractions and notations to specify concepts and semantics focused on, and usually restricted to, a particular problem domain. Several experience reports have demonstrated that DSLs can significantly improve the productivity to implement and deliver domain-specific concepts as the final software products [7, 8]. In academic, industrial and government communities, various DSLs have been proposed and used for describing, for example, 3D animations [9], business rules [10], insurance business logic [11], software testing [12] and military command and control [13].

This paper proposes a new model-driven framework that allows developers to model and program domain-specific concepts in DSLs and to transform them toward the final (compilable) source code in a seamless and piecemeal manner. Each DSL is defined as a UML metamodel extended from the standard UML metamodel. The proposed framework provides an abstraction to represent domain-specific concepts at both modeling and programming layers simultaneously. In the modeling layer, domain-specific concepts are represented as *Domain Specific Model (DSM)*, which is a set of UML diagrams compliant with a certain DSL (i.e. a UML metamodel). In the programming layer, domain-specific concepts are represented as *Domain Specific Code (DSC)*, which consists of program interfaces and *attributes* associated with them. Attributes are declarative *marks*, associated with program elements (e.g. interfaces and classes), to indicate that particular program elements maintain application-specific or domain-specific semantics [14]. The proposed framework transforms domain-specific concepts from modeling layer to programming layer, and vise versa, by providing a seamless mapping between DSMs and DSCs without any semantics loss.

The proposed framework transforms a DSM and DSC into a more detailed model and program by applying a given transformation rule. The framework allows developers to

* Work by Junichi Suzuki is supported by OGIS International, Inc. and Electric Power Development Co., Ltd.

define arbitrary transformation rules, each of which specifies how to specialize a DSM and DSC to particular implementation and/or deployment technologies. For example, a transformation rule may specialize them to a database system, while another one may specialize them to a business rule engine with a certain remoting support. Then, the proposed framework combines the specialized DSM and DSC and generates the final (compilable) source code.

This paper describes the design and implementation of the proposed framework, and discusses how the framework can improve the productivity to implement domain-specific concepts and how it can increase the longevity of models and programs representing domain-specific concepts. In order to demonstrate how to use the proposed framework, this paper also shows a development process using an example DSL to specify service-oriented distributed systems.

The structure of this paper is organized as follows. Section 2 summarizes the contributions of this work. Section 3 overviews attribute-oriented programming. Section 4 describes the design and implementation of the proposed framework, and Section 5 demonstrates how to use the proposed framework using an example DSL. Sections 6 and 7 conclude with comparison with existing related work and some discussion on future work.

2. CONTRIBUTIONS

This paper makes the following contributions to the design of model-driven development frameworks.

• ***UML2.0 support for modeling domain-specific concepts.*** The proposed framework accepts DSLs as metamodels extending the UML 2.0 standard metamodel, and uses UML 2.0 diagrams (class and composite structure diagrams) for modeling domain-specific concepts as DSMs. This work is the first attempt to leverage the UML2.0 model elements to define and use DSLs.

• ***Higher abstraction for programming domain-specific concepts.*** The proposed framework offers a new approach to represent domain-specific concepts at the programming layer, through employing the notion of attribute-oriented programming. This approach provides a higher abstraction to developers, and improves the productivity for them to program domain-specific concepts. Programming domain-specific concepts with attributes is much simpler and more readable than programming with general-purpose languages.

• ***Seamless mapping of domain-specific concepts between modeling and programming layers.*** The proposed framework maps domain-specific concepts between modeling and programming layers in a seamless and bi-directional manner. This mapping allows modelers[1] and programmers to deal with the same set of domain-specific concepts in different representations (i.e. UML models and program interfaces with attributes), yet at the same level of abstraction. This means that modelers do not have to involve the details of attribute-oriented programming and other programming responsibilities, and programmers do not have to possess detailed domain knowledge and UML modeling expertise. This separation of concerns can reduce the complexity in application development, and increase the productivity to model and program domain-specific concepts.

• ***Modeling layer support for program attributes.*** Through the bi-directional mapping between UML models and program attributes, the proposed framework provides a means to visualize program attributes (i.e. domain-specific concepts) as UML models. This paper presents the first attempt to bridge the gap between UML modeling and attribute-oriented programming.

3. BACKGROUND

Attribute-oriented programming is a program-level marking technique. Programmers can *mark* program elements (e.g. classes and methods) to indicate that they have application-specific or domain-specific semantics [14]. For example, some programmers may define a “logging” attribute and associate it with a method to indicate the method should implement a logging function, while other programmers may define a “web service” attribute and associate it with a class to indicate the class should be implemented as a web service. Attributes separate application’s core logic from application-specific or domain-specific semantics (e.g. logging and web service functions). By hiding the implementation details of those semantics from program code, attributes increase the level of programming abstraction and reduce programming complexity, resulting in simpler and more readable programs. The program elements associated with attributes are transformed to more detailed programs by a supporting tool (e.g. pre-processor). For example, a pre-processor may insert a logging program into the methods associated with a “logging” attribute.

The notion of attribute-oriented programming is becoming well accepted in several languages and tools, such as Java 2 standard edition (J2SE) 5.0 [15], C# [16] and XDoclet [17]. For example, J2SE 5.0 implements attributes as *annotations*, and the Enterprise Java Bean (EJB) 3.0 extensively uses annotations to make EJB programming easier [18]. Here is an example annotation in EJB 3.0

```
@entity class Customer{
  String name;
}
```

The `@entity` annotation is associated with the class `Customer`. This annotation indicates that the class `Customer` will be implemented as an entity bean in EJB. The pre-processor EJB provides, called *annotation processor*, takes an annotated code as an input and transforms it into final (compilable) code as an output. In this example, the annotation processor generates several interfaces and classes required to implement an entity bean (i.e. a remote interface, home interface and implementation class).

A transformation of annotated code is performed based on a certain transformation rule. The EJB annotation processor

[1] This paper assumes that modelers (or domain engineers) are familiar with particular domains but may not be programming experts.

follows the transformation rules predefined in the EJB 3.0 specification[2].

In addition to predefined annotations, J2SE 5.0 allows developers to define and use their own (i.e. user-defined) annotations. There are two types of user-defined annotations; *marker annotations* and *member annotations*. Here is an example marker annotation, named `Logging`.

```
public @interface Logging{ }
```

A marker annotation is defined with the keyword `@interface`.

```
public class Customer{
  @Logging public void setName(...){...}
}
```

In this example, the `Logging` annotation is associated with the method `setName()`, indicating that the method logs method invocations. Then, a developer who defines this `Logging` annotation specifies a transformation rule for the annotation, and creates a user-defined annotation processor that implements the transformation rule. The annotation processor may replace each annotated method with a method implementing a logging function[3].

A member annotation, the second type of user-defined annotations, is an annotation that has member variables.

```
public @interface Persistent{
  String connection();
  String tableName();
}
```

Here, the `Persistent` annotation has two member variables: `connection` and `tableName`.

```
@Persistent(
  connection = “jdbc:http://localhost/”,
  tableName = “customer”
)
public class Customer{}
```

The `Persistent` annotation is associated with the class `Customer`, indicating that the instances of `Customer` will be stored in a database with a particular database connection and table name. Then, a developer who defines this annotation specifies a transformation rule for the annotation, and implements a user-defined annotation processor that takes annotated code and generates additional classes implementing a database access function[3].

4. THE PROPOSED FRAMEWORK

This section describes the design and implementation of the proposed framework.

4.1 ARCHITECTURAL DESIGN

The proposed framework consists of two architectural components: DSC Generator and DSL Transformer (Figure 1). DSC Generator converts a DSM into a DSC, and vise versa. Each DSM consists of UML class diagrams and composite structure diagrams, which are compliant with a particular DSL. A DSL is a metamodel that extends the UML 2.0 standard metamodel with UML’s extension mechanism[4]. The UML extension mechanism provides a set of model elements such as *stereotype* and *tagged-value* in order to add application-specific or domain-specific modeling semantics to the standard UML metamodel [2]. Each DSL defines a set of stereotypes and tagged-values to express domain-specific concepts. Stereotypes are specified as metaclasses extending UML’s standard metaclasses, and tagged-values are specified as attributes of the extended metaclass. Given a DSL, a DSM represents domain-specific concepts using UML model elements associated with the stereotypes and tagged-values defined in the DSL.

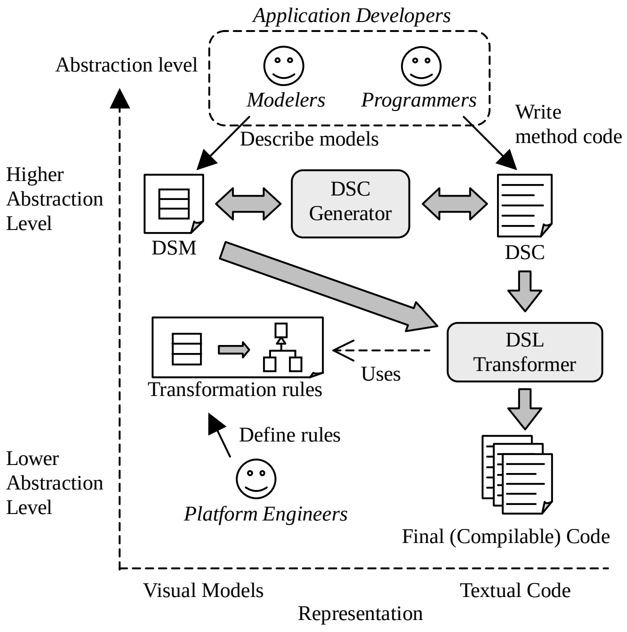

Figure 1: The architecture of the proposed framework

Each DSC consists of Java’s interfaces and/or classes decorated with the J2SE 5.0 annotations. The annotated code used in the proposed framework follows the J2SE 5.0 syntax to define marker and member annotations. DSC Generator transforms domain-specific concepts between modeling and programming layers by providing a direct one-to-one mapping between DSMs and DSCs.

The mapping between DSMs and DSCs allows modelers and programmers to deal with the same set of domain-specific concepts in different representations (i.e. as UML models and annotated code), yet at the same level of abstraction. This means that modelers do not have to involve the details of attribute-oriented programming and other programming responsibilities, and programmers do not have to know domain knowledge and UML modeling in

[2] The EJB 3.0 specification predefines a set of annotations and transformation rules for them.

[3] J2SE 5.0 provides a set of classes to help developers build their own (i.e. user-defined) annotation processors.

[4] A metamodel extending the standard UML metamodel is called a *UML profile* or *virtual metamodel* [2]. In a sense, each DSL is defined as a UML profile for the proposed framework.

detail. This separation of concerns can reduce the complexity in application development, and increase the productivity to model and program domain-specific concepts.

After DSC Generator generates a DSC, programmers write method code in the DSC (i.e. annotated code) in order to implement dynamic behaviors for domain-specific concepts (Figure 1). Please note that the methods of annotated code generated by DSC Generator are empty because both DSMs and DSCs specify the static structure of domain-specific concepts. Programming for annotated code is much simpler and more readable than programming for traditional program elements (e.g. interfaces and classes). Thus, the proposed framework provides a higher abstraction to developers, and improves the productivity for them to program domain-specific concepts.

DSL Transformer transforms DSM and DSC to the final (compilable) code by applying a given transformation rule (Figure 1). The proposed framework allows developers (platform engineers) to define arbitrary transformation rules, each of which specifies how to specialize DSMs and DSCs to particular implementation and/or deployment technologies (e.g. database and remoting technologies). Given a transformation rule, DSL Transformer first transforms (or unfolds) DSM model elements associated with stereotypes or tagged-values into plain UML model elements that do not have any stereotypes and tagged-values. In this transformation, a DSM is specialized to particular implementation and/or deployment technologies. DSL Transformer generates program code from the specialized DSM, and extracts method code maintained in DSC. Then, it produces the final compilable code by combining the generated program code and the extracted method code.

DSC Generator and DSL Transformer are separated by design. The proposed framework clearly separates the task to model and program domain-specific models (as DSMs and DSCs) from the task to transform them into the final compilable code. This design strategy improves separation of concerns between modelers/programmers and platform engineers. Modelers and programmers do not have to know how domain-specific concepts are implemented and deployed when modeling and programming them. Platform engineers do not have to know the details of domain-specific concepts. Also, modelers/programmers and platform engineers can perform their tasks in parallel. As a result, the proposed framework makes development process more streamlined and productive.

This design strategy also allows DSM/DSC and transformation rules to evolve independently, and contributes to increase the longevity of DSMs and DSCs. Since DSMs and DSCs do not depend on transformation rules, different transformation rules can be applied to a single set of DSM and DSC. This means that the proposed framework can specialize a single set of DSM and DSC to different implementation and deployment technologies by using different transformation rules. For example, a DSM and DSC may be specialized to Java RMI [15] first, SOAP [19] next, and then .NET remoting [20]. As such, the proposed framework can maintain domain-specific concepts (i.e. DSMs and DSCs) longer than the longevity of implementation and deployment technologies, thereby maximizing the reusability of domain-specific concepts.

4.2 MAPPING BETWEEN DSM AND DSC

The proposed framework maps DSM to DSC, vice versa, based on the following rules.

- A UML class in DSM is mapped to a Java class in DSC.
- A UML interface in DSM is mapped to a Java interface in DSC.
- A stereotype in DSM is mapped to a marker annotation in DSC.
- A tagged-value in DSM is mapped to a member annotation in DSC.

Figure 2 shows the class `Customer` stereotyped as <<`entitybean`>> with a tagged-value of `jndi-name="ejb/Customer"`. It is mapped to the following Java class, marker annotation and member annotation.

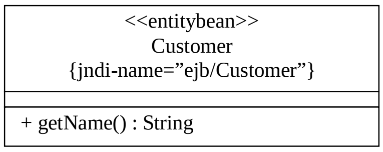

Figure 2: Class Customer

(1) Java class `Customer`

```
@entitybean
@jndi-name( value = "ejb/Customer" )
public class Customer{
        public String getName(){}
}
```

(2) Marker annotation `entitybean`

```
@interface entitybean{}
```

(3) Member annotation `jndi-name`

```
@interface jndi-name{
        string value();
}
```

4.3 DESIGN AND IMPLEMENTATION OF KEY COMPONENTS

This section describes the implementation of the proposed framework. As described in Section 4.1, the framework consists of two architectural components: DSC Generator and DSL Transformer. DSL Transformer is implemented with three components: DSM Transformer, Skeleton Code Generator and DSC Transformer (Figure 3). Every component in the proposed framework is implemented with Java.

DSC Generator: DSC Generator performs transformations between DSMs and DSCs (Figure 3). When accepting a DSM for a transformation, DSC Generator validates the input DSM against a corresponding DSL (i.e. metamodel)[5]. For example, it examines if the model elements in the input DSM use appropriate stereotypes and tagged-values defined

[5] Each DSL is defined as a metamodel extending the standard UML metamodel.

in a corresponding DSL. It also checks if they follow the semantics defined in the standard UML metamodel.

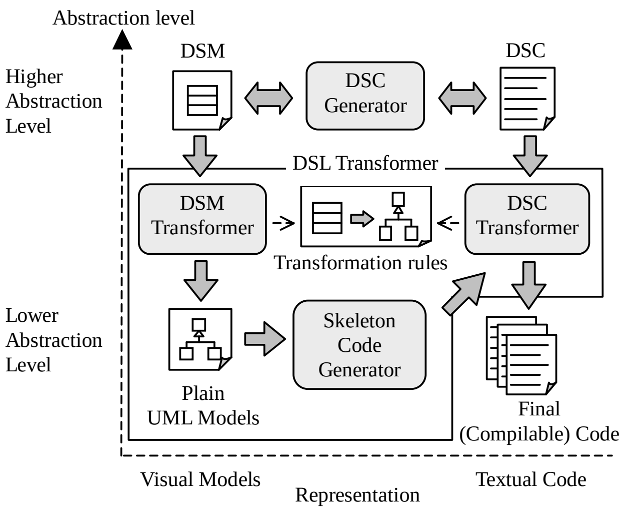

Figure 3: Key components in the proposed framework

Before accepting a DSM, the proposed framework needs to import a corresponding DSL. When importing a DSL, the framework validates the DSL. For example, it examines if the DSL correctly extends the UML standard metamodel.

DSC generator is implemented atop the Eclipse Modeling Framework (EMF)[6] and Eclipse-UML2[7]. The validation of DSMs and DSLs is implemented by extending the class `UML2Switch` provided by Eclipse-UML2. Once a DSM is validated, DCS Generator generates a DSC based on the mapping rules described in Section 4.2. The DSC generation is also implemented by extending the class `UML2Switch`.

In order to import DSMs and DSLs, the proposed framework accepts their representations in the XML Metadata Interchange (XMI) 2.0 [21]. XMI is an XML-based format to describe UML models. Developers can generate their DSMs or DSLs as XMI descriptions using any UML modeling tools that supports XMI 2.0. The following is the XMI representation of the class `Customer` in Figure 2.

```
<UML:Class
xmi.id="id_class" owner="id_project"
name="Customer" appliedSteotype=
"profile.xmi#//*
[@xmi.id=&quot;id_profile&quot;]">
<UML:Element.ownedElement>
 <UML:Operation xmi.id="id_operation"
  name="getName" owner="id_class">
  <UML:Element.ownedElement>
   <UML:Parameter
    xmi.id="id_param"
    type="id_operation"
    name="Unnamed" direction="result"
    owner="id_operation"/>
  </UML:Element.ownedElement>
 </UML:Operation>
 <UML:TaggedValue
  xmi.id="id_taggedvalue"
  name="jndi-name" owner="id_class">
  <UML:TaggedValue.dataValue>
   ejb/Customer
  </UML:TaggedValue.dataValue>
 </UML:TaggedValue>
</UML:Element.ownedElement>
</UML:Class>
<UML:DataType xmi.id="id_string"
owner="id_project" name="String"/>
```

The `<UML:Class>` tag defines a class, and its attribute `appliedStereotype` refers, with XPath directives, a stereotype defined in another XMI file (i.e. `profile.xmi`). The `<UML:TaggedValue>` tag defines a tagged-value associated with the class `Customer`.

DSM Transformer: DSM Transformer accepts a DSM and transforms it into more detailed models that specialize in particular implementation and/or deployment technologies (Figure 3). DSM Transformer performs this transformation in accordance with a transformation rule that a developer (platform engineer) defines (Figure 2).

DSM Transformer is implemented using the Model Transformation Framework (MTF)[8], which is implemented on EMF and Eclipse-UML2. MTF provides a language to declaratively define rules for transformations between EMF-based models. Each transformation rule consists of conditions and rules. DSM Transformer parses an input DSM to identify the model elements that meet the conditions, and applies the rules to them.

A transformation results in a set of plain UML models that do not have any stereotypes and tagged-values. The plain UML model specializes in particular implementation and/or deployment technologies. For example, if a transformation specializes an input DSM to Java RMI, the classes in the DSM are converted to the classes that implement the `java.rmi.Remote` interface.

Skeleton Code Generator: Skeleton Code Generator takes a plain UML model generated by DSM Transformer, and generates skeleton code in Java (Figure 3). The skeleton code is a Java representation of the input UML model. Since the proposed framework only supports structural UML diagrams (class and composite structure diagrams), the generated skeleton code does not have any code in methods. Skeleton Code Generator uses EMF and Eclipse-UML2 to accept and inspect input plain UML models.

DSC Transformer: DSC Transformer accepts the DSC generated by DSC Generator and the skeleton code generated by Skeleton Code Generator, and combines them to generate the final (compilable) code in Java. Using the Java reflection API, DSC Transformer extracts method code embedded in an input DSC[9], and copies the method code to an input skeleton code. DSC Transformer analyses a transformation rule, which is used to transform a DSM to a plain UML model, in order to determine where each method code is copied in an input skeleton code.

[6] http://www.eclipse.org/emf/

[7] http://www.eclipse.org/uml2/. Eclipse-UML2 implements the standard UML metamodel as a set of Java objects on EMF.

[8] http://www.alphaworks.ibm.com/tech/mtf/

[9] Note that programmers embed method code in the DSC generated by DSC Generator (Figure 1).

5. AN EXAMPLE DSL

This section describes an example DSL to describe domain-specific concepts in Service Oriented Architecture (SOA), and overviews a development process using the DSL with the proposed framework.

5.1 SOA DSL

SOA is a distributed systems architecture that connects network services in a platform independent manner [22]. In SOA, a service is a software component that has an interface accessible via network. A service's interface represents functions that the service provides. SOA hides details of underlying platform, e.g. service's implementation and remoting infrastructure (or middleware), and abstracts systems using two concepts, i.e. service's interface and connection between services. When developers create a distributed system using SOA, they (1) identify what kind of functions the system requires, (2) choose services that provide those required functions, and (3) connect those services to create a system. Connections between services require several methods to coordinate invocations (i.e. message exchanges) between services such as synchronizing several invocations, multicasting and conversing message formats. To provide such methods, mechanisms that abstract connections are required.

The proposed SOA DSL focuses on connectivity between services. It is defined as a UML profile, which provides model elements (stereotypes and tagged values) to define connections between services. The proposed DSL defines four types of elements, `Connector`, `Filter`, `Message` and `Service`. They are defined as stereotypes (Figure 4).

A `Service` represents a network service, and a `Message` represents a message exchanged between `Service`s. A `Connector` represents a connection between `Service`s. Developers can indicate invocation semantics such as synchronous invocation and asynchronous invocation using it (Figure 5). Also, `Connector` provides some functions, e.g. message encryption (Figure 5). A `Connector` can include arbitrary number of `Filter`s to define its behavior. A `Filter` represents a function such as message interceptor or multicast (Figure 6). A `Message` represents a data scheme that is exchanged between services via a `Connector`.

`Connector` is defined as a stereotype of `Class` metaclass in the `InternalStructures` package. The metaclass, an element defined in UML 2.0 composite structure diagram, can have internal structures such as class or interfaces. It allows developers to define nested structures in a visual manner, e.g. a class composed of several internal classes. The rest of defined stereotypes, i.e. `Service`, `Message` and `Filter`, are defined as a stereotype of Class metaclass in the Kernel package. It is a class in UML 2.0 class diagram.

`Connector` has two semantics, i.e. invocation semantics and connection semantics (Figure 5). Invocation semantics has three options, i.e. synchronous invocation, asynchronous invocation and oneway invocation. Connection semantics has four options, i.e. reliability, encryption, stream and queuing. Reliability option assures messages arrive to their destinations (i.e. services). Encryption option encrypts messages. Stream option enables streaming messaging. Queuing option deploys message queue in a connection. These options don't have any attributes such as resend interval, encryption algorithm and queue size. These configurations are defined in transformation rules from stereotypes to specific implementations.

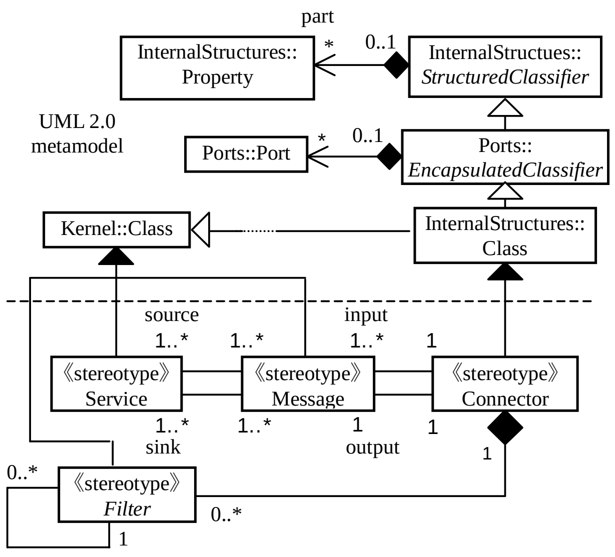

Figure 4: The proposed SOA DSL

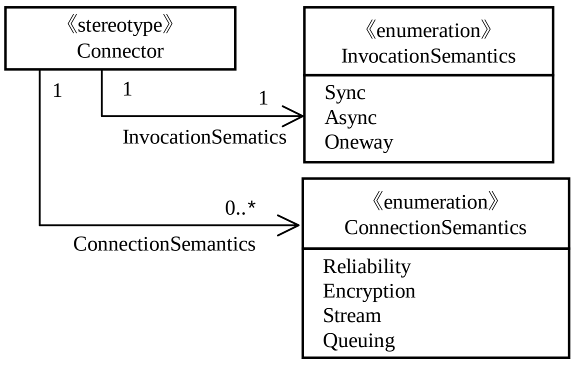

Figure 5: `Connector` stereotypes

`Filter` has four sub stereotypes, i.e. `MessageConverter`, `MessageAggregator`, `Multicast` and `Interceptor` (Figure 6). `MessageConverter` converts message scheme. `MessageAggregator` synchronizes multiple invocations and aggregates their messages. `Multicast` emits a message to several filters or services in a parallel manner. `Interceptor` hooks an invocation and examines its message. It allows developers to implement any kind of filters other than built-in ones.

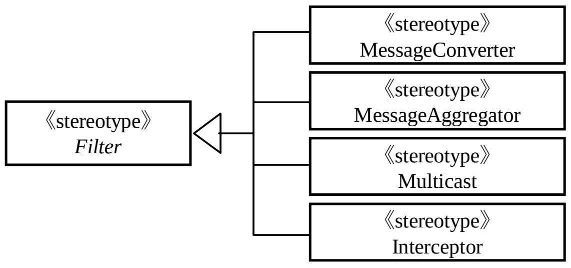

Figure 6: `Filter` stereotypes

5.2 DEVELOPMENT PROCESS USING THE PROPOSED FRAMEWORK AND SOA DSL

There are six key phases in an application development process using the proposed framework and SOA DSL.

(1) Define DSM. Modelers describe a DSM. A DSM is described in UML 2.0 class diagram or composite structure diagram, and UML profile. Figure 7 is an example model using SOA DSL. They represent concepts that are peculiar to targeted problem domain (i.e. a connector between services). In this model, there are three services, i.e. `Customer`, `Supervisor` and `Supplier`. `Customer` sends a message `OrderMessage`, `Supervisor` sends a message `Confirmation`, and `Supplier` receives a message `OrderMessage`. A connector `Connection` connects these services and delivers messages. Connection's invocation semantic is `Synchronize` and its connection semantic is `Encryption`. `Connection` has two filters, i.e. `Logger` and `Aggregator`. `Logger` is a class of `Interceptor`, and logs messages passes through this filter. `Aggregator` is a class of `MessageAggregator`. It aggregates two messages `OrderMessage` and `Confirmation`, and delivers `OrderMessage` to `Supplier`.

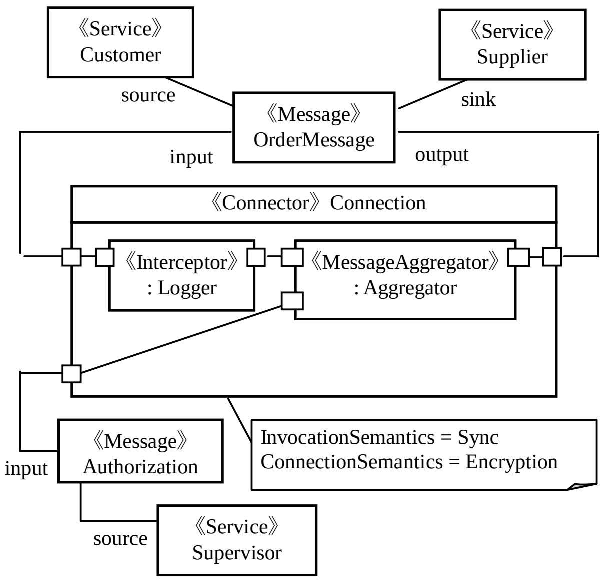

Figure 7: An example DSM using the proposed SOA DSL

(2) Transform DSM into DSM. The SOA DSL requires transforming DSM to DSM in order to specify which `Services` receive `Messages`. For example, the combination of `Supplier` and `OrderMessage`, i.e. `Supplier` receives `OrderMessage`, is transformed into the following DSL (Figure 8).

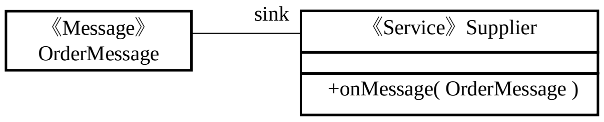

Figure 8: Transformed DSM

(3) Generate DSC. DSC Generator takes a DSM as an input and generates a DCS. The following DSC for `Supplier` is generated from the DSM (Figure 8).

```
@Service
public class Supplier{
public onMessage( OrderMessage message){} }
```

(4) Write Method Code. Programmers write method code on DSC in Java. For example, programmers write method code on `Supplier` class's `onMessage` method.

As described above, the abstraction levels of the two artifacts, i.e. DSM and DSC, are same. They employ different representations. DSM is written in UML and UML profiles. DSC is written in Java with annotations. UML profiles in DSM are converted into annotations in DSC. Application developers, i.e. modelers and programmers, can work at the same and high abstraction level. It makes the readability and maintainability of artifacts high, thereby high productivity can be archived.

(5) Define Transformation Rules. Platform engineers define transformation rules that are sets of model-to-model conversion rules. Each conversion rule converts UML model elements that have stereotypes or tagged-values into plain UML model. The plain UML model has details of underlying platform such as remoting middleware. For example, a transformation rule converts a UML class with stereotype `<<service>>` into several UML interfaces and classes that are necessary to use Java RMI (Figure 9) or SOAP when the corresponding connection supports synchronous invocation. Otherwise, it's transformed into several interfaces and classes that are necessary to use Java Message Service (JMS) [23] when the corresponding connection supports asynchronous invocation (Figure 9).

(6) Generate Final Code. DSL Transformer takes a DSM and a DSC as inputs, and generates final (compilable) code as following steps.

I. **Model-to-Model transformation.** According to transformation rules, DSL transformer converts a DSM into a plain UML model.

II. **Code Generation.** DSL Transformer generates skeleton code in Java from the plain UML models.

III. **Copy Method Code.** DSL Transformer extracts method code from DSC. According to transformation rules, it copies the method code to skeleton code generated at the previous step.

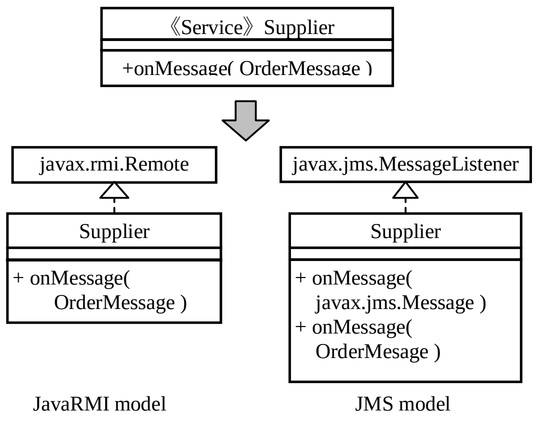
《Service》Supplier
+onMessage(OrderMessage)
javax.rmi.Remote
Supplier
+ onMessage(OrderMessage)
javax.jms.MessageListener
Supplier
+ onMessage(javax.jms.Message)
+ onMessage(OrderMesage)
JavaRMI model
JMS model

Figure 9: Service implementations with JavaRMI and JMS

6. RELATED WORK

The proposed framework reuses the J2SE 5.0 syntax to write annotated code (i.e. marker and member annotations). However, the proposed framework and J2SE 5.0 follow different approaches to define transformation rules between annotated code and compilable code. In J2SE 5.0, transformation rules are defined in a procedural manner (i.e. as programs) [15][10]. It allows developers to define arbitrary transformation rules in user-defined annotation processors that extend the default annotation processor (see Section 2). A user-defined annotation processor examines annotated code using Java's reflection API, and generates compliable code based on a corresponding transformation rule. Although this transformation mechanism is generic and extensible, it tends to be complicated and error-prone to write user-defined annotation processors. Also, transformation rules are not maintainable enough in annotation processors. When updating a transformation rule, a corresponding annotation processor needs to be modified and recompiled.

In contrast, the proposed framework allows developers to define transformation rules in a declarative manner (see Section 3.3). Declarative transformation rules are more readable and easier to write and maintain than procedural ones. It is not required to recompile the proposed framework when updating a transformation rule. Also, transformation rules are defined at the modeling layer, not the programming layer. This raises the level of abstraction for handling transformation rules, resulting in higher productivity for users to manage them.

XDoclet accepts declarative rules for transforming annotated code to compilable code [17]. In XDoclet, annotations are represented as comments in Java programs (Javadoc comments). Each transformation rule is defined as a template, which parameterizes an output program with variables representing the names of annotated program's elements (e.g. class names and method names). During a transformation, XDoclet invokes annotation handlers, which developers are required to write in Java for corresponding annotations. Each annotation handler examines annotated code using Java's reflection API, and generates output code by replacing template variables with the names of program elements gathered from annotated code. Declarative transformation rules are readable and easy to maintain in templates. However, similar to user-defined annotation processors in J2SE 5.0, it tends to be complicated to write and maintain annotation handlers. Developers need to keep maintaining the consistency between annotation handlers and templates (i.e. transformation rules). When updating a template, a corresponding annotation handler needs to be modified and recompiled.

Unlike XDoclet, the proposed framework requires developers to write nothing except declarative transformation rules. They are described at the modeling layer, not the programming layer. There is no need to recompile it when updating transformation rules. As such, it offers more productivity and less responsibility for developers to maintain transformation rules. Also, the proposed framework allows transformation rules to define output code compatible with a variety of programming languages (not only with Java), while XDoclet is limited to generate output code in Java.

The proposed framework has some functional commonality with existing model-driven development tools such as OptimalJ[11], Rose XDE[12], Together[13] and UMLX [24]. They are usually composed of two components: Model Transformer and Code Generator (Figure 10). Model Transformer converts (or unfolds) an UML model that modelers describe into plain UML model accordance with transformation rules. This phase lowers the abstraction level of an UML model. By doing this conversion process in an automatic and a traceable manner, model-driven development tools hide low-level details of targeted domains. However, programmers have to deal with generated code. Code Generator converts the plain UML models into code written in general-purpose programming languages such as Java. Since general-purpose languages can't represent domain concepts, each of them is described as one element in domain specific models, it needs to represent them using a set of elements (i.e. interfaces or classes) in generated (or unfolded) UML models and code. It complicates the generated UML models and code, and lowers their abstraction level. Programmers have to understand and deal with the low-level details of targeted domains even though model-driven development tools hide them. In contrast, the proposed framework provides DSC (i.e. annotated code) to represent targeted systems at higher abstraction in the programming layer (Figure 1). Programmers can work at the same high abstraction level of modelers. DSC can hide low-level details of targeted domain from not only modelers but also programmers. It makes the productivity high, especially for programmers.

[10] Transformation rules in .NET are also defined as in a procedural manner [18].

[11] http://www.compuware.com/products/optimalj/
[12] http://www.ibm.com/software/awdtools/developer/rosexde/
[13] http://www.borland.com/together/architect/

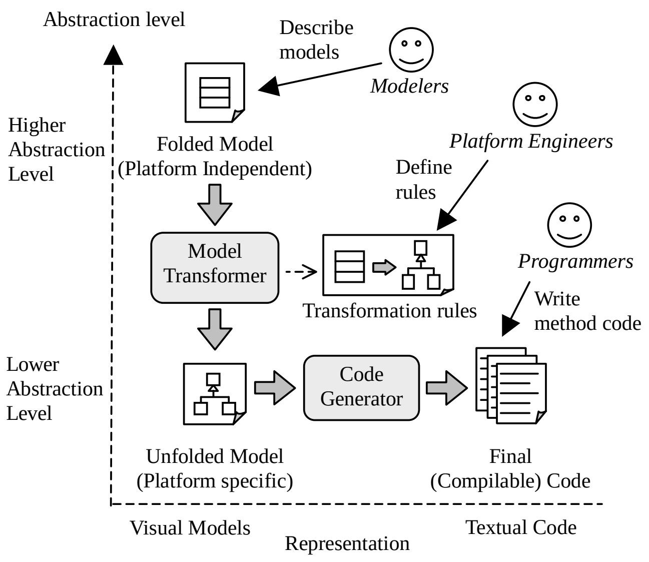

Figure 10: Development process using traditional model-driven development tools

Furthermore, the proposed framework better handles traceability issue between models and code. Traceability between models and code is one of the key issues in model driven development [25]. Ensuring the traceability, it needs to handle gaps between folded model and unfolded model, and unfolded model to code (Figure 10). Many existing model-driven development tools such as OptimalJ don't provide a reverse engineering function from code to model. If developers revise generated code, e.g. revising classes' structures, it can't ensure the consistency and the traceability between model and code. Some model-driven development tools such as Together can generate UML model from code. However it's a one-to-one mapping, therefore the abstraction level of generated UML model is same as compilable code. It's complicated and difficult to understand.

In contrast, the changes in code immediately feedback to folded model because the proposed framework assures a direct (i.e. one-to-one) mapping between DSM (i.e. UML model with profiles) and DSC (i.e. annotated code). The proposed framework can ensure the consistency and traceability between model and code (i.e. from model to code, and code to model).

Model Transformation Framework (MTF) is a tool that helps developers make comparisons, check consistency, and implement transformations between Eclipse Modeling Framework (EMF) models. It focuses on model transformation, doesn't generate code. Kent Modeling Framework provides a language for model transformation, Yet Another Transformation Language (YATL) [26]. It supports UML's Object Constraint Language (OCL) [27] to define transformation rules. It's able to define transformation rules to transform UML models to specific programming languages such as Java. However, similar to other model-driven development tools, generated source code is unfolded and complicated. In contrast, the proposed framework generates high abstract code using annotations.

Several model-driven development tools such as BridgePoint[14], iUML and iCCG [28], and SMART [29] support executable UML models. These tools can run and debug a UML model, and generate compilable code. Defining an executable model, developers write actions (e.g. data processing, method invocations) using an action language. Each tool has its own action language, e.g. Object Action Language for BridgePoint, Action Specification Language for iUML, and SMART Action Language for SMART, because there is no standard of the syntax of action language. UML standardizes the semantics of action language, i.e. action semantics specification [2], but it doesn't have concrete syntax. Therefore, developers are required to learn these proprietary languages. It makes the productivity low. In contrast, developers can use any programming languages when using the proposed framework. It doesn't require developers to learn new language, and lowers the learning curve.

7. CONCLUDING REMARKS

This paper proposes a new framework that allows developers to model and program domain-specific concepts with DSLs and to transform them toward the final (compilable) source code in a model-driven manner. The proposed framework provides an abstraction to represent domain-specific concepts at both modeling and programming layers by leveraging the notions of UML metamodeling and attribute-oriented programming. This paper presents the development process using the proposed framework as well as several key designs in the framework, and describes how the framework can improve the productivity to implement domain-specific concepts and increase the longevity of models and code representing domain-specific concepts. As an example to show how the proposed framework handles DSLs, this paper also presents a DSL used to define service-oriented distributed system architectures.

Several extensions to the proposed framework are planned as future work. The proposed framework currently supports only one DSL for each transformation from DSM to compilable code. A future work will address generating compilable code through combining DSMs and DSCs written in multiple DSLs. For example, one of the authors has been building a DSL to express insurance claims processing. A future experiment will have the proposed framework accept the insurance DSL as well as the SOA DSL described in this paper, in order to evaluate the impact of using multiple DSLs simultaneously on the framework design.

The framework is also being extended to support the .NET remoting [21] in addition to JMS and RMI so that it can generate the source code compatible with broader range of implementation technologies. Another extension is to support a standardized language for transformation rules, such as the MOF Query/Views/Transformations (MOF QVT) specification [30], which is currently standardized in the Object Management Group.

[14]http://www.acceleratedtechnology.com/embedded/nuc_modeling.html

ACKNOWLEDGEMENT

The authors would like to thank Adam Malinowski for his valuable comments that improved the quality of this paper.

8. REFERENCE

[1] B. Selic, "The Pragmatics of Model-Driven Development" In *IEEE Software*, vol. 20, no. 5, September/October, 2003.

[2] Object Management Group, *UML 2.0 Superstructure Specification*, http://www.omg.org/, 2004.

[3] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," In *IEEE Software*, vol. 20, no. 5, Sept./Oct. 2003.

[4] G. Booch, A Brown, S Iyengar, J. Rumbaugh, and B. Selic, "An MDA Manifesto," In *D. Frankel and J. Parodi (ed.), The MDA Journal: Model Driven Architecture Straight from the Masters*, Chapter 11, Meghan-Kiffer Press, Dec. 2004.

[5] S. Cook, "Domain-Specific Modeling and Model-driven Architecture," In *D. Frankel and J. Parodi (ed.), The MDA Journal: Model Driven Architecture Straight from the Masters*, Chapter 3, Meghan-Kiffer Press, Dec. 2004.

[6] A. van Deursen, P. Klint and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," In *ACM SIGPLAN Notices*, vol. 35, no. 6: 26-36, 2000.

[7] S. Kelly and J. Tolvanen, "Visual Domain-specific Modeling: Benefits and Experiences of using metaCASE Tools," In *Proc. of International workshop on Model Engineering, ECOOP 2000*.

[8] R. Kieburtz et al., "A Software Engineering Experiment in Software Component Generation," In *Proc. of 18th IEEE International Conference on Software Engineering*, 1996.

[9] C. Elliott, "Modeling Interactive 3D and Multimedia Animation with an Embedded Language," In *Proc. of First USENIX Conference on Domain-Specific Languages*, Oct. 1997.

[10] G. Wagner, S. Tabet, and H. Boley, "MOF-RuleML: The Abstract Syntax of RuleML as a MOF Model", In *Proc. of* Integrate 2003, Oct. 2003.

[11] H. Wegener, "Balancing Simplicity and Expressiveness: Designing Domain-Specific Models for the Reinsurance Industry", In *Proc. of the 4th OOPSLA Wrokshop on Domain-Specific Modeling*, Vancouver, BC, Canada, Oct. 2004.

[12] Object Management Group, *UML Testing Profile*, 2004. http://www.omg.org/

[13] D. Wile, "Lessons Learned from Real DSL Experiments," In *Proc. of the 36th Hawaii International Conference on System Sciences*, 2003.

[14] D. Schwarz, "Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5", In *ON Java.com*, O'Reilly Media, Inc., June 2004.

[15] Sun Microsystems, *Java 2 Platform, Standard Edition 5.0*, 2004. http://java.sun.com/

[16] ISO/IEC, *C# Language Specification*, Chapter 24: Attributes, ISO/IEC 23270, 2003.

[17] C. Walls and N. Richards, *XDoclet in Action*, Manning Publications, December 2003.

[18] Sun Microsystems, *Enterprise Java Beans 3.0 Early Draft Review*, 2004. http://java.sun.com/

[19] W3C, *SOAP Version 1.2*, 2003. http://www.w3.org/

[20] R. Wiener, "Remoting in C# and .NET," In *Journal of Object Technology*, vol. 3, no. 1, January-February 2004.

[21] Object Management Group, *MOF 2.0 XML Metadata Interchange*, http://www.omg.org/, 2004.

[22] L. Baresi, R. Heckel, S. Thöne and D. Varró, "Modeling and Validation of Service-Oriented Architectures: Application vs. Style," In *Proc. of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2003.

[23] Sun Microsystems, *Java Messaging Service Specification Version 1.1*. http://java.sun.com/

[24] E. Willink, "UMLX: A Graphical Transformation Language for MDA", In *Proc. of OOPSLA '02*, November 2002.

[25] J. Champeau and E. Rochefort, "Model Engineering and Traceability," In *Proc. of <<UML>> 2003*, October 2003.

[26] O. Patrascoiu, "Mapping EDOC to Web Services using YATL," In *Proc. of the 8th IEEE International Enterprise Distributed Object Computing Conference*, September 2004.

[27] Object Management Group, *UML 2.0 Object Constraint Language*, 2004. http://www.omg.org/

[28] C. Raistrick, P. Francis and J. Wright, *Model Driven Architecture with Executable UML*, Cambridge University Press, March 2004.

[29] S. Hayashi, P. Yibing, M. Sato, K. Mori, S. Sejeon and S. Haruna, "Test Driven Development of UML Models with SMART Modeling System," In *Proc. of <<UML>> 2004*, October 2004.

[30] Object Management Group, *MOF 2.0 Query / Views / Transformations*, 2004. http://www.omg.org/