A Model Transformation Framework for Domain Specific Languages: An Approach Using UML and Attribute-Oriented Programming[*]

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 business or technology domain) and to transform them toward the final (compilable) source code. The proposed framework provides an abstraction to represent domain-specific concepts at both modeling and programming layers by leveraging the notions of UML and attribute-oriented programming. At the modeling layer, domain-specific concepts are represented as a *Domain Specific Model (DSM)*, which is a set of UML 2.0 diagrams. At the programming layer, domain-specific concepts are represented as a *Domain Specific Code (DSC)*, which consists of program elements and *attributes* associated with them. The proposed framework transforms domain-specific concepts from the modeling layer to programming layer, and vise versa, by providing a seamless mapping between DSMs and DSCs. The proposed framework transforms a DSM and DSC into a more detailed model and program that specialize in a particular implementation and deployment technologies. Finally, the 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.

Keywords: Model Driven Development, Domain Specific Language, UML, Attribute-Oriented Programming

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 implementation technology domains toward the concepts and semantics in 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-specific concepts.

Domain Specific Language (DSL) is a promising solution to directly capture, represent and implement domain-specific concepts [4]. 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 [5]. In academic, industrial and government communities, various DSLs have been proposed and used for describing, for example, 3D animations [6], business rules [7], insurance business logic [8], software testing [9] and military command and control [10].

This paper proposes a new model-driven development 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. By leveraging the notions of UML metamodeling and attribute-oriented programming, the proposed framework provides an abstraction to represent domain-specific concepts at the modeling and programming layers simultaneously. At the modeling layer, domain-specific concepts are represented as a *Domain Specific Model (DSM)*, which is a set of UML 2.0 diagrams compliant with a DSL. Each DSL is defined as a UML metamodel that extends the UML 2.0 standard (superstructure) metamodel[1]. At the programming layer, domain-specific concepts are represented as a *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 [11]. The proposed framework transforms domain-specific concepts from the modeling layer to program-

[*] The work by Junichi Suzuki is supported by OGIS International, Inc. and Electric Power Development Co., Ltd.

[1] This work is the one of first attempt to leverage the UML2.0 model elements to define and use DSLs.

ming 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 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 rule 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.

The structure of this paper is organized as follows. Section 2 overviews attribute-oriented programming. Section 3 describes the design and implementation of the proposed framework. Sections 4 and 5 conclude with comparison with existing related work and some discussion on future work.

2. ATTRIBUTE-ORIENTED PROGRAMMING

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 [11]. 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, such as Java 2 standard edition (J2SE) 5.0 [12] and C# [13]. 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 [14]. 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 preprocessor 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 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].

---

[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.

3. THE PROPOSED FRAMEWORK

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

In the proposed framework, a DSL is defined as 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 metaclasses (i.e. stereotypes).

Given a DSL, a DSM represents domain-specific concepts using the UML 2.0 model elements associated with the stereotypes and tagged-values defined in the DSL. Each DSM consists of UML class diagrams and composite structure diagrams.

Each DSC consists of Java interfaces and 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.

3.1 MAPPING BETWEEN DSM AND DSC

The proposed framework transforms DSMs to DSCs, vice versa, based on the following mapping rules.

- A UML package in DSM is mapped to a Java package in DSC.
- 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 method of UML classes/interfaces in DSM is mapped to a method of Java classes/interfaces in DSC.
- A stereotype that does not have tagged-values in DSM is mapped to a marker annotation.
- A stereotype that has tagged-values in DSM is mapped to a member annotation.
- A tagged-value in DSM is mapped to a member variable of a member annotation in DSC.
- A property of UML classes in DSM is mapped to a property (data field) of Java classes in DSC.
- A modifier and visibility of UML elements in DSM is mapped to a modifier and visibility of Java elements in DSC.
- A UML primitive type in DMS is mapped to a Java type in DSC.

Figure 1 shows the class `Customer` stereotyped as `<<entitybean>>` with a tagged-value of `jndi-name="ejb/Customer"`. Based on the above mapping rules, the proposed framework transforms the UML class (DSM) to the following Java class and member annotation.

[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.

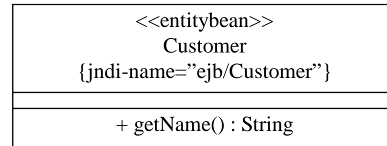

Figure 1: UML Class Customer (DSM)

(1) Java class `Customer (DSC)`

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

(2) Member annotation `entitybean (DSC)`

```
@interface entitybean{
  String jndi-name();
}
```

The proposed framework transforms domain-specific concepts between the modeling and programming layers in a seamless and bi-directional manner. It allows modelers[5] 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[6]. 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 detail. This separation of concerns can reduce the complexity in application development, and increase the productivity to model and program domain-specific concepts.

3.2 DESIGN AND IMPLEMENTATION OF KEY COMPONENTS

The proposed framework consists of two architectural components: DSC Generator and DSL Transformer (Figure 2). DSL Transformer is implemented with three components: DSM Transformer, Skeleton Code Generator and DSC Transformer (Figure 2). Every component in the proposed framework is implemented with Java.

DSC Generator: DSC Generator performs transformations between DSMs and DSCs following the mapping rules described section 3.1 (Figure 2). When accepting a DSM for a transformation, DSC Generator imports a DSL that the input DSM follows, and validates the input DSM against the imported DSL. For example, it examines if the model elements in the input DSM use appropriate stereotypes and tagged-values defined in the DSL. It also checks if they follow the semantics defined in the standard UML metamodel.

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

[6] Through the bi-directional mapping between UML models (DSMs) and program attributes (DSCs), the proposed framework provides a means to visualize program attributes in UML models. This paper presents the first attempt to bridge the gap between UML modeling and attribute-oriented programming.

In order to import DSMs and DSLs, the proposed framework accepts their representations in the XML Metadata Interchange (XMI) 2.0 [16]. XMI is an XML-based format to describe UML models. Developers can generate XMI descriptions from their DSMs and DSLs using any UML modeling tools that supports XMI 2.0. The following is the XMI representation of the class `Customer` shown in Figure 1.

```
<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` references, with XPath directives, a stereotype defined in another XMI file (`profile.xmi`). In this example, the stereotype `entitybean` is referenced with its identifier `id_profile`. The `<UML:TaggedValue>` tag defines a tagged-value associated with the class `Customer`.

DSC generator is implemented with the Eclipse Modeling Framework (EMF)[7], Eclipse-UML2[8] and the Eclipse Java Development Tool (JDT)[9]. It imports DSMs as XMI descriptions and parses them to construct their in-memory tree structures, called UML trees, with Eclipse-UML2. The validation of DSMs is implemented by extending the class `UML2Switch` provided by Eclipse-UML2. Once a DSM is validated, DCS Generator traverses model elements in a UML tree using Eclipse-UML2, and constructs a Java Abstract Syntax Tree (JAST) that corresponds to the UML tree. The JAST is built with JDT based on the mapping rules described in Section 3.1. Finally, DSC Generator generates a DSC (i.e. annotated Java code) from the JAST.

After DSC Generator generates a DSC, programmers write method code in the generated DSC (i.e. annotated code) in order to implement dynamic behaviors for domain-specific concepts[10]. Since annotated code is simpler and more readable than traditional (unannotated) program code, programming for annotated code is less complex than programming without annotations. Therefore, the proposed framework improves the productivity for developers to program domain-specific concepts.

DSM Transformer: DSM Transformer accepts a DSM and transforms it into more detailed models. In accordance with a transformation rule that a developer (platform engineer)[11] defines, DSM Transformer 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 (Figure 2). 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.

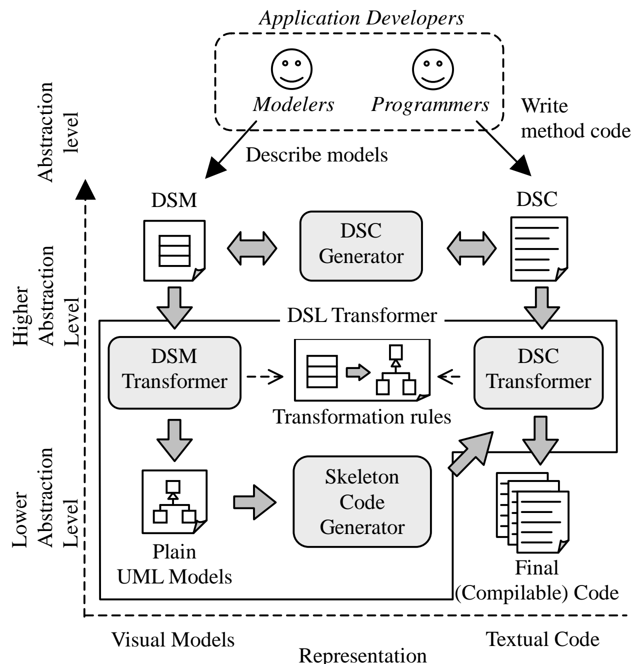

Figure 2: Key components in the proposed framework

DSM Transformer is implemented using the Model Transformation Framework (MTF)[12], 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.

Skeleton Code Generator: Skeleton Code Generator takes a plain UML model generated by DSM Transformer, and gener-

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

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

[9] http://www.eclipse.org/jdt/

[10] Please note that the methods in DSC are empty because both DSMs and DSCs only specify the static structure of domain-specific concepts (a DSM consists of class and composite structure diagrams).

[11] This paper assumes that platform engineers possess the expertise in platform technologies on which DSMs and DSCs are deployed. They are responsible to define transformation rules between models.

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

ates skeleton code in Java (Figure 2). 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. DSC Transformer extracts method code embedded in an input DSC, 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.

By separating DSC Generator and DSL Transformer, 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 [12] first, and SOAP [15] next. 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. 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) [12][13]. 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. 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.

The proposed framework has some functional commonality with existing model-driven development tools such as OptimalJ[14], Rose XDE[15], Together[16], UMLX [17] and Kent Modeling Framework [18]. They usually have two functional components: Model Transformer and Code Generator (Figure 3). Similar to DSM Transformer in the proposed framework, Model Transformer accepts UML models that modelers describe with UML profiles, and converts them to more detailed models in accordance with transformation rules. Similar to Skeleton Code Generator in the proposed framework, Code Generator takes the UML models created by Model Transformer, and generates source code.

A major difference between existing model-driven development tools and the proposed framework is the level of abstraction where programmers work. In existing model-driven develop-

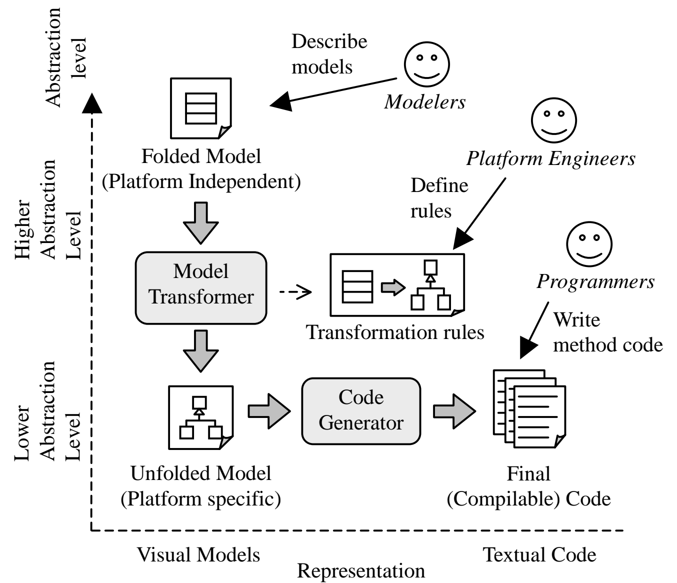

Figure 3: Development process using traditional model-driven development tools

[13] Transformation rules in .NET are also defined as in a procedural manner [13].

[14] http://www.compuware.com/products/optimalj/

[15] http://www.ibm.com/software/awdtools/developer/rosexde/

[16] http://www.borland.com/together/architect/

ment tools, programmers and modelers work at different abstraction levels (Figure 3). Although modelers work on UML modeling at a higher abstraction level, programmers need to handle the source code, at a lower abstraction level, which is generated by Code Generator. (Figure3). The generated source code is often hard to read and understand. It tends to be complicated, time consuming and error-prone to modify and extend the source code.

Unlike the existing model-driven development tools, the proposed framework allows both programmers and modelers to work at a higher abstraction level (Figure 2). Programmers implement behavioral functionalities (i.e. method code) in DSCs before DSL Transformer transforms them to more detailed programs that specialize in particular implementation and deployment technologies. This means that programmers can focus on coding application's core logic (or business logic) without handling the details in implementation and deployment technologies. Also, the DSCs (i.e. annotated code) generated by DSC generator are much more readable and easier to understand than the source code generated by Code Generator in the existing model-driven development tools (Sections 2 and 3.1). Therefore, the proposed framework provides a higher productivity for programmers to implement their applications.

Furthermore, the proposed framework better ensures the consistency between domain-specific models and code. Many existing model-driven development tools such as OptimalJ do not provide a reverse engineering function from code to model. If developers revise generated code, e.g. revising classes' structures, it can not ensure the consistency between model and code. Some model-driven development tools such as Together can generate UML model from code. However they ensure the consistency between code and unfolded models, not domain-specific models (UML models with UML profile). Existing model-driven development tools enforce their own development process, i.e. revising domain-specific models first and implementing code, for developers.

In contrast, in the proposed framework, the changes in code immediately feedback to domain-specific models because the level of abstraction of DSM and DSC is the same, and the proposed framework assures a direct mapping between DSM and DSC. Developers can choose which representation, i.e. at the modeling layer and at the programming layer, they handle. The proposed framework helps much kind of development processes than existing model-development tools.

5. 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.

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 a DSL for a certain implementation technology, in order to evaluate the impact of using multiple DSLs simultaneously on the framework design.

6. 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] 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.

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

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

[7] 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.

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

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

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

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

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

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

[14] Sun Microsystems, *Enterprise Java Beans 3.0 Early Draft Review 2*, 2005. http://java.sun.com/

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

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

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

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