Modeling Turnpike Frontend System: a Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming*

Hiroshi Wada and Junichi Suzuki

Department of Computer Science
University of Massachusetts, Boston
hiroshi_wada@otij.org and jxs@cs.umb.edu

Abstract. This paper describes and empirically evaluates a new model-driven development framework, called Modeling Turnpike (or mTurnpike). It allows developers to model and program domain-specific concepts (ideas and mechanisms specific to a particular business or technology domain) and to transform them to the final (compilable) source code. By leveraging UML metamodeling and attribute-oriented programming, mTurnpike provides an abstraction to represent domain-specific concepts at the modeling and programming layers simultaneously. The mTurnpike frontend system transforms domain-specific concepts from the modeling layer to programming layer, and vise versa, in a seamless manner. Its backend system combines domain-specific models and programs, and transforms them to the final (compilable) source code. This paper focuses on the frontend system of mTurnpike, and describes its design, implementation and performance implications. In order to demonstrate how to exploit mTurnpike in application development, this paper also shows a development process using an example DSL (domain specific language) to specify service-oriented distributed systems.

1. Introduction

Modeling technologies have matured to the point where they can offer significant leverage in all aspects of software development. 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 [1].

Domain Specific Language (DSL) is a promising solution to directly capture, represent and implement domain-specific concepts [1, 2]. DSLs are the languages targeted to particular problem domains, rather than general-purpose languages that are aimed at any software problems. Several experience reports have demonstrated that DSLs can improve the productivity in implementing domain-specific concepts [3].

This paper proposes a new model-driven development framework, called Modeling Turnpike (or mTurnpike), which aids modeling and programming domain-specific concepts with DSLs. mTurnpike allows developers to model and program domain-

* Research supported in part by OGIS International, Inc. and Electric Power Development Co., Ltd.

specific concepts in DSLs and to transform them to the final (compilable) source code in a seamless and piecemeal manner. Leveraging UML metamodeling and attribute-oriented programming, mTurnpike 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 described in a DSL. Each DSL is defined as a UML metamodel that extends the UML 2.0 standard metamodel [4]. At the programming layer, domain-specific concepts are represented as a *Domain Specific Code (DSC)*, which consists of attribute-oriented programs. Attributes are declarative *marks*, associated with program elements (e.g. classes and interfaces), to indicate that the program elements maintain application-specific or domain-specific semantics [5]. The frontend system of mTurnpike transforms domain-specific concepts from the modeling layer to programming layer, and vise versa, by providing a seamless mapping between DSMs and DSCs without any semantics loss.

The backend system of mTurnpike transforms a DSM and DSC into a more detailed model and program by applying a given transformation rule. mTurnpike allows developers to define arbitrary transformation rules, each of which specifies how to specialize a DSM and DSC to particular implementation and deployment technologies. For example, a transformation rule may specialize them to a database, while another rule may specialize them to a remoting system. mTurnpike combines the specialized DSM and DSC to generate the final (compilable) source code.

This paper focuses on the frontend system of mTurnpike, and describes its design, implementation and performance implications. In order to demonstrate how to exploit mTurnpike in application development, this paper also shows a development process using an example DSL to specify service-oriented distributed systems.

2. Contributions

This section summarizes the contributions of this work.

- ***UML 2.0 support for modeling domain-specific concepts.*** mTurnpike accepts DSLs as metamodels extending the UML 2.0 standard metamodel, and uses UML 2.0 diagrams to model domain-specific concepts (as DSMs). This work is one of the first attempts to exploit UML 2.0 to define and use DSLs.
- ***Higher abstraction for programming domain-specific concepts.*** mTurnpike offers a new approach to represent domain-specific concepts at the programming layer, through the notion of attribute-oriented programming. This approach provides a higher abstraction for developers to program domain-specific concepts, thereby improving their programming productivity. Attribute-oriented programming makes programs simpler and more readable than traditional programming paradigms.
- ***Seamless mapping of domain-specific concepts between the modeling and programming layers.*** mTurnpike maps domain-specific concepts between the 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 attribute-oriented programs), yet at the same level of abstraction. Thus, modelers do not have to involve programming details, and programmers do not have to possess

[1] This paper assumes modelers are familiar with particular domains but may not be programming experts.

detailed domain knowledge and UML modeling expertise. This separation of concerns can reduce the complexity in application development, and increase the productivity of developers in modeling and programming domain-specific concepts.

- ***Modeling layer support for attribute-oriented programs.*** Using the bi-directional mapping between UML models and attribute-oriented programs, mTurnpike visualizes attribute-oriented programs in UML. This work is the first attempt to bridge a gap between UML modeling and attribute-oriented programming.

3. Background: 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 maintain application-specific or domain-specific semantics [5]. For example, a programmer may define a "logging" attribute and associate it with a method to indicate the method should implement a logging function, while another programmer 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 (business) 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 has been well accepted in several languages and tools, such as Java 2 standard edition (J2SE) 5.0, C# and XDoclet[2]. For example, J2SE 5.0 implements attributes as *annotations*, and the Enterprise Java Beans (EJB) 3.0 extensively uses annotations to make EJB programming simpler. Here is an example using an EJB 3.0 annotation.

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

The `@entity` annotation is associated with the class `Customer`. This annotation indicates that `Customer` will be implemented as an entity bean. A pre-processor in EJB, called *annotation processor*, takes the above annotated code and applies a certain transformation rule to generate several interfaces and classes required to implement `Customer` as an entity bean (i.e. remote interface, home interface and implementation class). The EJB annotation processor follows the transformation rules predefined in the EJB 3.0 specification.

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

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

In J2SE 5.0, a marker annotation is defined with the keyword `@interface`.

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

[2] http://xdoclet.sourceforge.net/

In this example, the `Logging` annotation is associated with `setName()`, indicating the method logs method invocations. Then, a developer 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.

A member annotation, the second type of user-defined annotations, is an annotation that has member variables. It is also defined with `@interface`.

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

The `Persistent` annotation has the `connection` and `tableName` variables.

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

Here, the `Persistent` annotation is associated with the class `Customer`, indicating the instances of `Customer` will be stored in a database with a particular database connection and table name. A developer who defines this annotation implements a user-defined annotation processor that takes an annotated code and generates additional classes and/or methods implementing a database access function.

4. Design and Implementation of mTurnpike

mTurnpike consists of the frontend and backend systems (Fig. 1). The frontend system is implemented as DSC Generator, and the backend system is implemented as DSL Transformer. Every component in mTurnpike is implemented with Java.

The frontend system transforms domain-specific concepts from the modeling layer to programming layer, and vise versa, by providing a seamless mapping between DSMs and DSCs. In mTurnpike, a DSL is defined as a metamodel that extends the UML 2.0 standard (superstructure) metamodel with UML's extension mechanism[3]. 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 UML 2.0 standard metamodel [6]. In mTurnpike, 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 properties of stereotypes (i.e. extended metaclasses).

Given a DSL, a DSM is represented as a set of UML 2.0 diagrams (class and composite structure diagrams). Each DSC consists of Java interfaces and classes decorated with the J2SE 5.0 annotations. The annotated code follows the J2SE 5.0 syntax to define marker and member annotations.

The backend system of mTurnpike transforms a DSM and DSC into a more detailed model and program that specialize in particular implementation and deployment technologies. Then, it combines the specialized DSM and DSC to generate the final (compilable) code (Fig. 1).

In mTurnpike, the frontend and backend systems are separated by design. mTurnpike clearly separates the task to model and program domain-specific models

[3] An extended metamodel is called a *UML profile*. Each DSL is defined as a UML profile in mTurnpike.

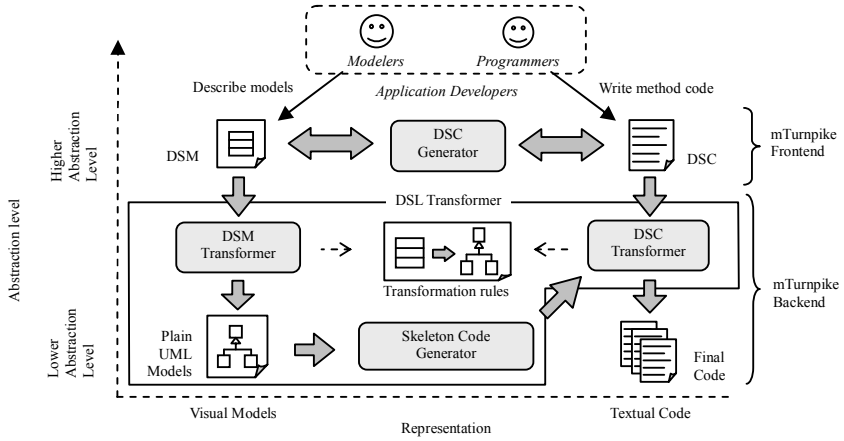

Fig. 1. mTurnpike Architecture and its Key Components.

(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[4]. Modelers and programmers do not have to know how domain-specific concepts are implemented and deployed in detail. Platform engineers do not have to know the details of domain-specific concepts. As a result, mTurnpike can reduce the complexity in application development, and increase the productivity of developers in modeling and programming domain-specific concepts.

This design strategy also allows DSMs/DSCs and transformation rules to evolve independently. Since DSMs and DSCs do not depend on transformation rules, mTurnpike can specialize a single set of DSM and DSC to different implementation and deployment technologies by using different transformation rules. When it comes time to change a running application, modelers/programmers make the changes in the application's DSM and DSC and leave transformation rules alone. When retargeting an application to a different implementation and/or deployment technology, e.g. Java RMI to Java Messaging Service (JMS), platform engineers define (or select) a transformation rule for the new target technology and regenerate the final compilable source code. As such, mTurnpike can make domain-specific concepts (i.e. DSMs and DSCs) more reusable and extend their longevity, thereby improving productivity and maintainability in application development.

4.1. Mapping between DSMs and DSCs in the mTurnpike Fontend System

mTurnpike implements the mapping rules shown in Table 1 to transform DSMs to DSCs, and vice versa. Fig. 2 shows an example DSM, the class `Customer` stereotyped as `<<entitybean>>` with a tagged-value. mTurnpike transforms the UML class (DSM) to the following Java class and member annotation (DSC).

[4] Platform engineers possess expertise in platform technologies on which DSMs and DSCs are deployed. They are responsible for defining transformation rules applied to DSMs and DSCs.

Table 1. Mapping rules between DSMs and DSCs.

UML Elements in DSM		Java Elements in DSC
DSL or Profile (M2)	Definition of a stereotype that has no tagged-values	Definition of a marker annotation
	Definition of a stereotype that has tagged-values	Definition of a member annotation
	Definition of a tagged-value	Definition of a member variable in a member annotation
DSM (M1)	Package	Package
	Class and interface	Class and interface
	Method and data field	Method and data field
	Modifier and visibility	Modifier and visibility
	Primitive type	Primitive type
	Stereotype that has no tagged-values	Marker annotation
	Stereotype that has tagged-values	Member annotation
	Tagged-value	Member annotation's member variable

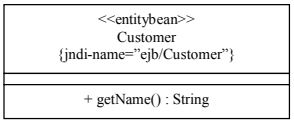

Fig. 2. UML Class `Customer` (DSM)

(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(); }
```

4.2. Design and Implementation of the mTurnpike Frontend System

The mTurnpike frontend system is implemented by DSC Generator (Fig. 1). It performs transformations between DSMs and DSCs based on the mapping rules described in Section 4.1. The following five steps involve in the transformation.

(1) Loading a DSM to build a UML tree: DSC Generator imports a DSM as a representation of the XML Metadata Interchange (XMI) 2.0 [7]. Developers can generate XMI descriptions of their DSMs using any UML tools that support XMI 2.0. Here is an example XMI description showing the class `Customer` in Fig. 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_string"
    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`.

When accepting a DSM, DSC Generator identifies a DSL that the input DSM follows. DSLs are also represented as XMI descriptions. In the above example, a DSL is defined in a file named `profile.xmi`. DSC Generator parses a DSM and its corresponding DSL, as XMI files, to build an in-memory tree structure, called UML tree. A UML tree is an instance of the UML standard metamodel. For building UML trees, DSC Generator follows the data structures provided by the Eclipse Modeling Framework (EMF)[5] and Eclipse-UML2[6].

Once a UML tree is constructed, DSC Generator validates the UML tree (i.e. an input DSM) against the standard UML metamodel. It examines if the DSM follows the syntax and semantics defined in the standard UML metamodel. DSC Generator also validates the UML tree (i.e. an input DSM) against a corresponding DSL. For example, it checks if the DSM uses appropriate stereotypes and tagged-values defined in the DSL. The validation of UML trees (DSMs) is performed by traversing the trees using a visitor class, named `UML2Switch` provided by Eclipse-UML2.

(2) Building a JAST for a DSL: Once a UML tree is built and validated, DSC Generator constructs a Java Abstract Syntax Tree (JAST) corresponding to a DSL represented in the UML tree. DSC Generator traverses a UML tree, using `UML2Switch` in Eclipse-UML2, and constructs a JAST node corresponding to each node in the UML tree based on the mapping rules described in Section 4.1.

Fig. 3 shows some key data structures to construct JASTs. `Annotation` represents an annotation. In order to represent a member annotation, `Annotation` has an association with `AnnotationMembers`, each of which represents its member variable. `AnnotationMember` keeps a value of member variable. `AnnotationDefinition` and `AnnotationMemberDefinition` represent

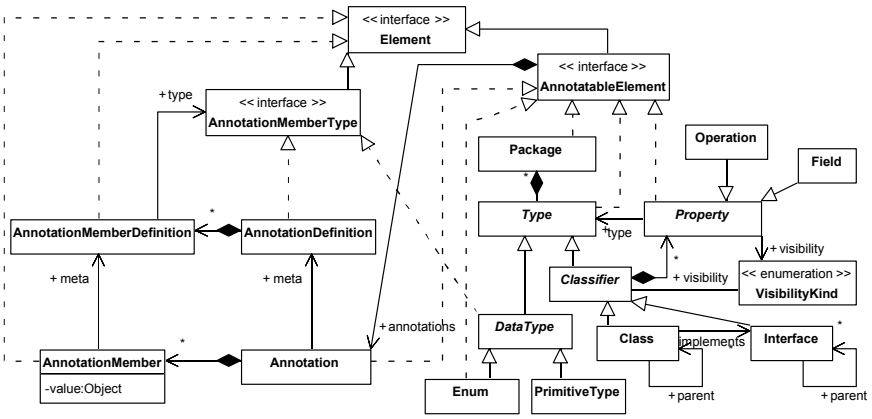

Fig. 3. Key data structures to construct Java Abstract Syntax Trees

[5] www.eclipse.org/emf/

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

the definitions of an annotation and an annotation's member variable. They are powertypes `Annotation` and `AnnotationMember`, respectively.

(3) Building a JAST for a DSC: After constructing a JAST corresponding to a DSL represented in a UML tree, DSC Generator completes the JAST by transforming the rest of the UML tree into JAST nodes. Transformations are performed with the JAST data structures shown in Fig. 3, following the mapping rules described in Section 4.1. In Fig. 3, `AnnotatableElement` is the root interface for the Java program elements that can be decorated by J2SE 5.0 annotations.

The following code fragment shows how DSC Generator transforms a stereotyped UML class (i.e. a class in DSM) to an annotated Java class (i.e. a class in DSC). The method `convertClass()` takes a UML class and instantiates the class `Class` in a JAST, which represents a Java class (see also Fig. 3). Then, the method transforms the stereotypes applied to the UML class to Java annotations by instantiating the class `Annotation` in `resolveStereotypes()` and `convertStereotype()`.

```
import edu.umb.cs.dssg.mturnpike.java.ast.*;
Class convertClass( org.eclipse.uml2.Class c_ ) {
  Class c = new Class();     // create a Java class as a JAST node
  resolveStereotypes(c, c_);// create a Java annotation(s), if a UML class is stereotyped.
  return c;
}
void resolveStereotypes( AnnotatableElement annotatableElement,
org.eclipse.uml2.Element element ) {
  foreach( Stereotype s in element.getAppliedStereotypes() ){
    Annotation annot = convertStereotype( element, s );
    annotatableElement.addAnnotation( annot ); }
}
Annotation convertStereotype( org.eclipse.uml2.Element element,
org.eclipse.uml2.Stereotype stereotype ) {
  Annotation annotation = new Annotation();
  String name = stereotype.getName();
  annotation.setName( name );
  AnnotationDefinition annotDefinition = getAnnotDefinition( name );
  annotation.setMeta( annotationDefinition );
  foreach( Property p in stereotype.getAttributes() ){
    AnnotationMember annotMember = new AnnotationMember();
    ... // set the name, type and definition of the created annotation member.
    annotation.addMember(annotationMember); }
  return annotation;
}
```

(4) Building a DSC (annotation definitions): Once a JAST is constructed, DSC Generator generates annotation definitions in a DSC. Each JAST node has the `toString()` method, which generates Java source code corresponding to the JAST node. DSC Generator traverses a JAST and calls the method on instances of `AnnotationDefinition` and `AnnotationMemberDefinition` (Fig. 3).

(5) Building a DSC: Once generating annotation definitions, DSC Generator generates the rest of annotated code in a DSC. DSC Generator traverses a JAST and calls the `toString()` method on each node in the JAST.

After DSC Generator generates a DSC (i.e. annotated code), programmers write method code in the generated DSC in order to implement dynamic behaviors for domain-specific concepts[7]. Please note that the methods in the generated DSC are

[7] 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).

empty because DSMs specify only the static structure of domain-specific concepts (using UML class diagrams and composite structure diagrams).

In addition to transformations from DSMs to DSCs, mTurnpike can perform reverse transformations from DSCs to DSMs. In a reverse transformation, mTrunpike parses a DSC (i.e. annotated Java code) with a J2SE 1.5 lexical analyzer (J2SE 1.5 parser) [8], and builds a JAST following the data structure shown in Fig. 3. The JAST is transformed to a UML tree and an XMI file using Eclipse-UML2

4.3. Design and Implementation of the mTurnpike Backend System

The mTurnpike backend system consists of three components: DSM Transformer, Skeleton Code Generator and DSC Transformer (Fig. 1).

DSM Transformer: DSM Transformer accepts a DSM as a UML tree built by DSC Generator, and transforms it to a more detailed model (Fig. 1). Given a transformation rule that a platform engineer defines, DSM Transformer transforms (or unfolds) DSM model elements associated with stereotypes and 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 deployment technologies. For example, if a transformation specializes an input DSM to Java RMI, the classes in the DSM are converted to the classes implementing the `java.rmi.Remote` interface.

DSM Transformer is implemented with the Model Transformation Framework (MTF) [9], which is implemented on EMF and Eclipse-UML2. MTF provides a language to define transformation rules between EMF-based models. mTurnpike follows the syntax of MTF's transformation rule language to specialize DSMs. Each transformation rule consists of conditions and instructions. DSM Transformer traverses a DSM (i.e. a UML tree built by DSC Generator), identifies the DSM model elements that meet transformation conditions, and applies transformation instructions to them. This process generates another UML tree that represents a model specializing in particular implementation and deployment technologies. The following is an example transformation rule.

```
relate class2class(
  uml:Class src when equals(match over src.stereotypes.name, "Service"),
  uml:Class tgt,
  uml:Interface tgt2 when equals(tgt2.name, "Remote")
  ) when equals(src.name, tgt.name){
       implementation(tgt, tgt2)
}
relate implementation(uml:Class c1, uml:Interface c2){
       realize(over c1.implementation, c2)
}
relate realize(uml:Implementation i, uml:Interface c){
       check interfaces(g.contract, c)
}
relate interfaces(uml:Interface c1, uml:Interface c2)
  when equals(c1.name, c2.name)
```

The keyword `relate` is used to define a transformation rule. This example defines four transformation rules. Each rule accepts model elements as parameters and instructs how to transform them. For example, the first rule (`class2class`) accepts the classes stereotyped with `<<Service>>`, and transform each of them to two

[8] mTurnpike's lexical analyzer is implemented with JavaCC (http://javacc.dev.java.net/).
[9] http://www.alphaworks.ibm.com/tech/mtf/

classes. One of the two classes has the same name as an input `<<Service>>` class, and it extends an interface whose name is `Remote`.

Skeleton Code Generator: Skeleton Code Generator takes a UML tree created by DSM Transformer, and generates skeleton code in Java (Fig. 1). It traverses an input UML tree, builds a JAST corresponding to the UML tree using the JAST package shown in Fig. 3, and generates Java code from the JAST. Since the mTurnpike frontend system only supports structural UML diagrams (class and composite structure diagrams), the methods in the generated code are empty.

DSC Transformer: DSC Transformer accepts a DSC generated by DSC Generator, method code written on the generated DSC by programmers, and skeleton code generated by Skeleton Code Generator. Then, DSC Transformer 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 by DSM Transformer, in order to determine where each method code is copied in an input skeleton code.

5. An Example DSL

This section describes an example DSL to develop service-oriented distributed systems, and overviews a development process using the DSL with mTurnpike.

5.1. SOA DSL

Service Oriented Architecture (SOA) is a distributed systems architecture that connects and operates network services in a platform independent manner. SOA models a distributed system as a collection of services. It abstracts distributed systems using two concepts, *service interface* and *connections between services*, and hides the details of implementation and deployment technologies, such as programming languages used to implement services and remoting infrastructures used to operate services. In SOA, each service maintains its own interface that makes its functionality accessible to other services via network. Each connection between services is an abstraction to specify how to interact (or exchange messages) between services.

The proposed SOA DSL focuses on connectivity between services, and allows developers to visually design the connections between services as UML diagrams. It

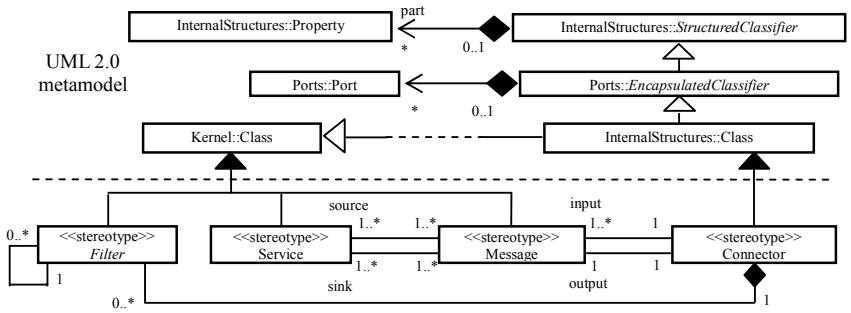

Fig. 4. The proposed SOA DSL

is defined as a UML profile extending the standard UML metamodel. Fig. 4 shows the core part of the proposed SOA DSL.

`Service` and `Message` are stereotypes used to specify network services and messages exchanged between services. They are defined as a metaclass extending the `Class` metaclass in the standard UML metamodel.

`Connector` is used to represent connections between services. It is a stereotype extending the `Class` metaclass in the `InternalStructures` package (Fig 4). This metaclass defines a model element used in the UML composite structure diagram. It allows developers to define nested model structures, such as a class composed of several internal (nested) classes. `Connector` maintains two different semantics: *connection semantics* and *invocation semantics* (Fig. 5). `ConnectionSemantics` is used to specify how to establish a connection between services. It defines four different semantics (Fig. 5). The `Reliability` option guarantees that messages are delivered to destinations. The `Encryption` option instructs that messages are encrypted on a connection. The `Stream` option enables streaming messages. The `Queuing` option deploys a message queue between services to enable a store-and-forward messaging policy. `InvocationSemantics` is used to specify how to invoke a service through a connection. Supported invocation semantics include synchronous, asynchronous and oneway invocations.

A Connector can contain Filters to customize its behavior (Fig. 4). The proposed SOA DSL currently defines four different filters (Fig. 6). MessageConverter converts the schema of messages exchanged on a connection. MessageAggregator synchronizes multiple invocations and aggregates their messages. Multicast simultaneously sends out a message to multiple filters or services. Interceptor is a hook to intercept invocations and examine messages.

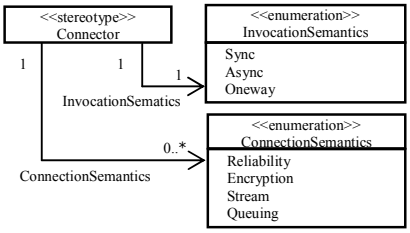

Fig. 5. `Connector` stereotypes

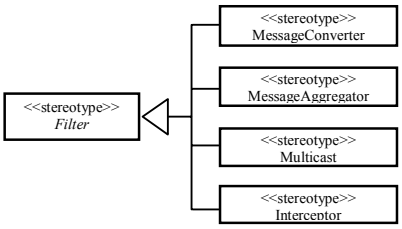

Fig. 6. `Filter` stereotypes

5.2. Development Process Using mTurnpike and SOA DSL

Using a SOA DSL described in Section 5.1, this section overviews an application development process with mTurnpike.

(1) Defining a DSM. Modelers define a DSM in the UML 2.0 class diagrams or composite structure diagrams. Fig. 7 shows an example DSM using the SOA DSL described in Section 5.1. `Customer` orders a product to `Supplier` by sends out an `OrderMessage`. If a `Supervisor` approves the order by issuing an `Authorization`, `Aggregator` aggregates the `OrderMessage` and `Authorization`, and sends an aggregated message to `Supplier`. `Connection`

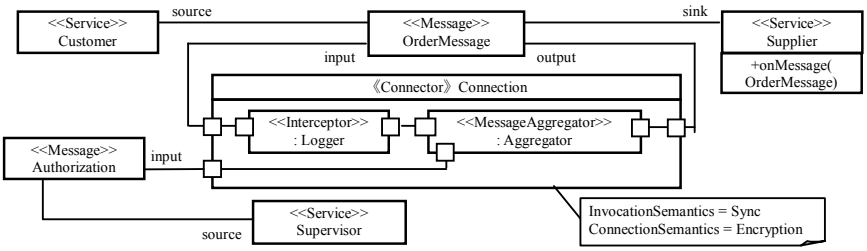

Fig. 7. An example DSM using the proposed SOA DSL

is responsible for connecting services and delivering messages between them. It establishes a synchronous and secure connection between services.

(2) Generating a DSC from a DSM. DSC Generator takes a DSM and generates a DCS (Fig. 1). The following is a DSC (annotated code) for `Supplier`.

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

(3) Writing Method Code. Programmers write method code in the generated DSC in Java (Fig. 1). For example, they write `onMessage()` in the `Supplier` class.

(4) Defining Transformation Rules. Platform engineers define a transformation rule to specialize a DSM in particular implementation and deployment technologies (Fig. 1). For example, if a DSM specifies a synchronous connection, a transformation rule may transform a UML class stereotyped with `<<service>>` into several UML interfaces and classes that are required to implement the `<<service>>` class as a Java RMI object (Fig. 8). If a DSM specifies an asynchronous connection, a transformation rule may specialize the `<<service>>` class to a JMS object (Fig. 8). The transformation rule also may specialize a `<<Message>>` class (e.g. `OrderMessage`) to implement the interface `javax.jms.Message`.

(5) Generate Final Code. DSL Transformer takes a DSM and a DSC as inputs, and generates the final (compilable) code (Fig. 1). It applies a transformation rule described in the step (4) to an input DSM to specialize the DSM. Then, it generates skeleton code in Java from the specialized DSM. Finally, DSL Transformer extracts method code from a DSC, and copies the method code to the generated skeleton code.

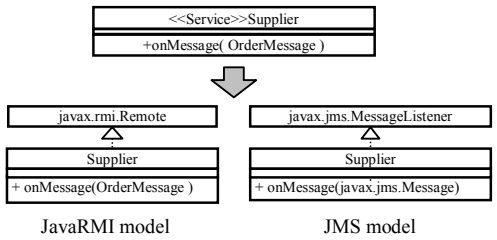

Fig. 8. Service implementations with JavaRMI and JMS

6. Preliminary Performance Evaluation

This section empirically evaluates the efficiency and memory footprint of the mTurnpike frontend system. Measurements are obtained with nine configurations (Table 2). For example, in the A1 configuration, mTurnpike loads a DSM that contains 10 classes, each of which has five data fields, one stereotype and five tagged-values (120 model elements in total). Measurements uses a Sun J2SE 5.0.2 VM running on a Windows 2000 PC with an Athlon 1.7 Ghz CPU and 512MB memory.

Table 2. Measurement configurations.

The number of model elements defined in each class	The number of classes		
	10	100	500
5 data fields and 1 stereotype (5 tagged-values for each stereotype)	A1 (120)	B1 (1200)	C1 (6000)
10 data fields and 2 stereotypes (5 tagged-values for each stereotype)	A2 (230)	B2 (2300)	C2 (11500)
50 data fields and 10 stereotypes (5 tagged-values for each stereotype)	A3 (1110)	B3 (11100)	C3 (55500)

In order to evaluate the efficiency of the mTurnpike frontend system, Fig. 9 shows the time for mTurnpike to execute each of the five functional steps to transform a DSM to a DSC (see Section 4.2). The numbers placed in the figure depicts how long it takes for mTurnpike to execute functional steps 1, 3 and 5. Fig. 9 shows mTurnpike is efficient enough in the configurations A1 to B2 (its overhead is up to 5 seconds). The transformation overhead is acceptable in small-scale to mid-scale application development. mTurnpike does not interrupt developers' modeling and programming work severely. Fig. 9 also shows that it takes 8 up to 33 seconds for mTurnpike to execute its frontend process in the C1 to C3 configurations. Several optimization efforts are currently underway, and they are expected to reduce the latency.

In order to examine the memory footprint of the mTurnpike frontend system, Fig. 10 shows how much memory space mTurnpike consumes to transform a DSM to a DSC. mTurnpike consumes no more than 15MB memory to handle models produced in small-scale up to large-scale projects (in the configurations A1 to C2). Since the memory utilization of mTurnpike is fairly small, it is not necessary for developers to upgrade their development environments (e.g. memory modules in their PCs).

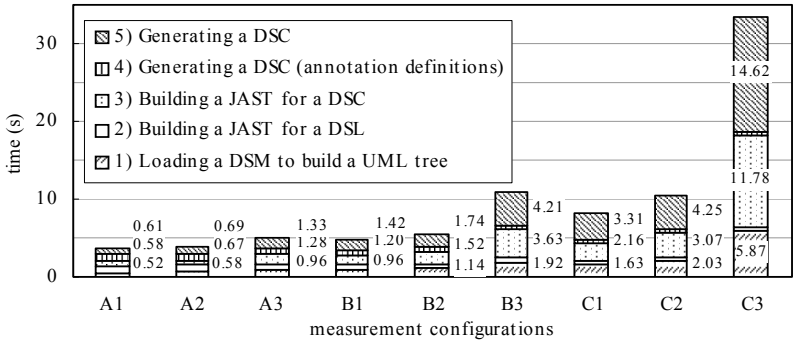

Fig. 9. Overhead of mTurnpike to transform a DSM to a DSC.

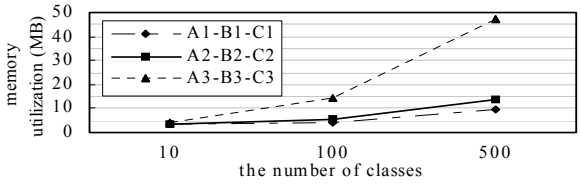

Fig. 10. Memory footprint of mTurnpike to transform a DSM to a DSC.

7. Related Work

mTurnpike reuses the J2SE 5.0 syntax to write annotated code (i.e. marker and member annotations). However, mTurnpike 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). It allows developers to define arbitrary transformation rules in user-defined annotation processors (see Section 2). A user-defined annotation processor examines annotated code using the Java reflection API, and generates compilable 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 difficult to maintain in annotation processors, since updating a transformation rule requires modifying and recompiling the corresponding annotation processor.

In contrast, mTurnpike allows developers to define transformation rules in a declarative manner. Declarative transformation rules are more readable and easier to maintain than procedural ones. It is not required to recompile mTurnpike when updating transformation rules. 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 of users in managing them.

mTurnpike has some functional commonality with existing model-driven development (MDD) tools such as OptimalJ[10], Rose XDE[11], Together[12], UMLX [8] and KMF [9] They usually have two functional components: Model Transformer and Code Generator (Fig. 11). Similar to DSM Transformer in mTurnpike, 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 mTurnpike, Code Generator takes the UML models created by Model Transformer, and generates source code.

A major difference between existing MDD tools and mTurnpike is the level of abstraction where programmers work. In existing MDD tools, programmers and modelers work at different abstraction levels (Fig. 11). Although modelers work on UML modeling at a higher abstraction level, programmers need to handle source code, at a lower abstraction level, which is generated by Code Generator (Fig. 11). 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.

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

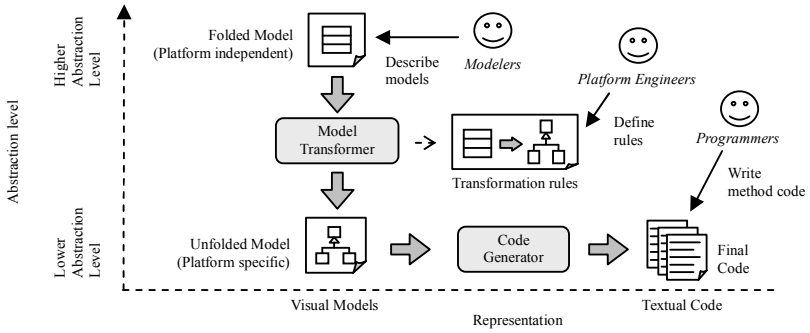

Fig. 11. Development process using traditional model-driven development tools.

Unlike existing MDD tools, mTurnpike allows both programmers and modelers to work at a higher abstraction level (Fig. 1). Programmers implement behavioral functionalities (i.e. method code) in DSCs, before DSL Transformer transforms DSCs 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, DSCs (i.e. annotated code) are much more readable and easier to maintain than the source code generated by Code Generators in existing MDD tools (see Sections 2 and 3.1). Therefore, mTurnpike provides a higher productivity of programmers in implementing their applications.

8. Conclusion

This paper describes and empirically evaluates a new model-driven development framework called mTurnpike. In addition to an overview of architectural design, this paper focuses on the frontend system of mTurnpike and describes its design, implementation and performance implications. In order to demonstrate how to exploit mTurnpike in application development, this paper also shows a development process using an example DSL to specify service-oriented distributed systems.

References

1. G. Booch, A Brown, S Iyengar, J. Rumbaugh and B. Selic, "An MDA Manifesto," In *The MDA Journal: Model Driven Architecture Straight from the Masters*, Chapter 11, Meghan-Kiffer Press, December 2004.
2. S. Cook, "Domain-Specific Modeling and Model-driven Architecture," In *The MDA Journal: Model Driven Architecture Straight from the Masters*, Chapter 3, Meghan-Kiffer Press, December 2004.
3. 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.
4. Object Management Group, *UML 2.0 Superstructure Specification*, Otober, 2004.
5. D. Schwarz, "Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5," In *ON Java.com*, O'Reilly Media, Inc., June 2004.
6. L. Fuentes, A. Vallecillo. "An Introduction to UML Profiles". *UPGRADE, The European Journal for the Informatics Professional,* 5 (2): 5-13, April 2004.
7. Object Management Group, *MOF 2.0 XML Metadata Interchange*, 2004.
8. E. Willink, "UMLX: A Graphical Transformation Language for MDA," In *Proc. of OOPSLA*, 2002.
9. O. Patrascoiu, "Mapping EDOC to Web Services using YATL," In *Proc. of the 8th IEEE International Enterprise Distributed Object Computing Conference*, September 2004.