Tool support for generating implementations of MOF-based modeling languages

Lutz Bichler
Institute for Software Technology
University of the German Federal Armed Forces Munich
85577 Neubiberg, Germany

Abstract

The upcoming version 2.0 of the UML standard will be the basis for many domain specific modeling languages. Therefore, tool support for implementing support for domain specific languages has to be built based on the MOF 2.0 standard, which defines the language used to specify the UML. Because manually implementing tools for domain specific modeling languages is time-consuming, generative tools are needed, which are flexible enough to be used in a variety of use cases, such as implementing repositories, case tools, model compilers a.s.o.

In this paper we present a part of the MOF Meta-modeling Tools (MOmo) project, which aims to create tools which support the MOF 2.0 standard. The MOmo toolbox will contain tools for defining and compiling MOF 2.0 models. Additionally, the tools are extensible for other MOF-based modeling languages such as UML and UML profiles. Within this paper we focus on the compiler component and describe the single processing steps which are carried out to generate an implementation from a model definition.

1 Introduction

The upcoming version 2.0 of the Unified Modeling Language (UML) ([7], [8]) will be the basis for many future domain specific modeling languages. UML is specified by a metamodel which defines the properties of the model elements. Extensions and adaptations will be done by redefining existing model elements and by adding new elements to the metamodel. The UML 2.0 meta-model is an instance of the *Meta Object Facility (MOF) 2.0* ([1]) meta-modeling language.

Modeling languages need to be tool supported in order to be really useful. Tool development is time and cost intensive. Therefore it needs to be supported by tools which facilitate the development of tools which implement the specific modeling languages. One of these "meta-tools" is the *MOF Meta-model Compiler* (MOmoC) which is presented in the following section. The standard configuration

of MOmoC reads XMI representations of MOF models and generates implementation code. In the following section we describe the compilation process, present a brief example and provide an overview over the MOmoC implementation.

2 Transformation Steps

Figure 1 provides an overview over the steps which are carried out to transform an XMI representation into implementation code. The first step is to read a MOF-model and build an object representation. This object representation can be modified by user modules in order to prepare it for a specific target language. The adapted object representation is used to build an internal XML-representation of the MOF-model. Finally, the internal representation is transformed into target language code.

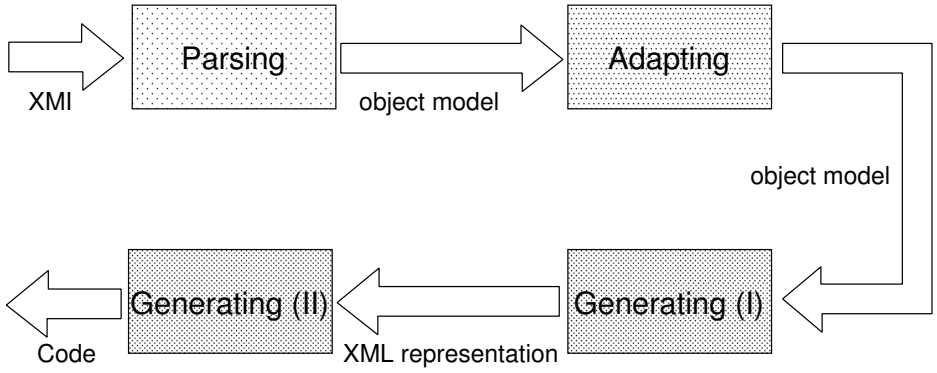

Figure 1: Transformation steps

The storage format for MOF-models is standardized by the OMG in the XMI specification ([6]). Therefore MOmoC processes an XMI file to generate implementation code for the contained model. As XMI was designed to be a flexible representation of model data in XML, the representation of object data in XMI is configurable. This resulted in many slightly different XMI dialects. Thus, in order to be able use XMI as input we need to transform it into one specific XMI-configuration or to use another representation of the MOF-model internally.

We decided to use an own XML representation of a MOF model, which is better suited for code generation than XMI. The main difference between XMI and our XML representations is that XMI in most cases contains the XML representations of several model elements, while our internal representation contains one XML document per model element.

The mapping from the object representation to the internal XML representation is straightforward. Each model element is mapped to an XML document

which contains a root node with the name of the model element. Each reference is mapped to an XML node with the name of the referenced object which contains the reference. In section 3 we show an example for the mapping of object representation to XML representation. Additionally to the mappings of the single objects a document for the model is created. This document contains references to the root level objects of the model and can be used as starting point for targets which exist only once per model, such as an interface for reading and writing XMI.

This representation has two main advantages over processing XMI directly. It is easier to read than XMI, because each document contains only a small part of the model and secondly, and it facilitates the use of *Extensible Stylesheet Language Transformations (XSLT)* ([9]) to transform it to implementation code for the same reason.

XSLT is a language to transform XML documents to other representations which is standardized by the W3C. Using XSLT for the transformation has several advantages. XSLT is a standard, it is well documented and many implementations are available. The basics are easy to learn and therefore simple changes to adapt the backend to a specific use case are straightforward. On the other hand XSLT is powerful enough to be used for complex code generation tasks.

Our internal XML-format mitigates the most important disadvantages of XSLT with regard to readability and performance. XSLT is often critisied for its verbosity, which makes stylesheets hard to read and understand. But the complexity of an XSLT stylesheet depends on the complexity of the input file. Our XML-documents are quite simple and therefore processable by relatively simple stylesheets. Although this does not avoid the verbosity of XSLT, it makes the stylesheets easier to read and write.

Additionally the breaking-up of the XML-representation into one document per model element increases the performance of the XSLT processing, because only the parts relevant for code generating need to be processed. Additionally the performance loss compared to backends which are written in a programming language can be descreased by compiling the XSLT stylesheets to Java classes.

3 Example

This section shows an example from the MOF 1.4 specification, which shows the steps of the generation process carried out by the MOmoC. The processing starts with an XMI-file, which contains the representation of the MOF model. At first the XMI file is transformed into an object representation and afterwards into the XML representation.

```
<?xml version="1.0" encoding="ISO8859_1"?>
<class name="Classifier" namespace="Model">
   <superclasses>
      <classref href="Model.GeneralizableElement"
                isParent="true"
                isInherited="true"/>
      <classref href="Model.Namespace"
                isParent="false"
                isInherited="true"/>
      <classref href="Model.ModelElement"
                isParent="false"
                isInherited="true"/>
   </superclasses>
   <ownedAttributes/>
   <ownedOperations/>
   <subclasses/>
</class>
```

Figure 2: Internal representation of MOF 1.4 meta-class Classifier in XML

Figure 2 shows the representation of the meta-class Classifier from the MOF 1.4 specification in the internal XML format. The document contains a root node <class> which contains subnodes for the properties of the class. The <superclasses> node contains sub-nodes that reference each inherited class. By two attributes, isParent and isInherited, directly inherited classes are separated from the classes which are indirectly inherited. This facilitates the mapping of the multiple inheritance of MOF 2.0 to programming languages which only provide single inheritance.

The internal representation is transformed into target language code by applying stylesheets. Figure 3 shows a cut-out from the stylesheet which transforms class representations into nsuml-compatible Java interfaces. It is shown that XSLT templates are used to generate the implementation code. The content of the templates is Java code mixed with XSLT processing instructions. The Java code defines a template which is specific for a certain type of modeling element. The XSLT code is responsible for filling in the parts which are specific for each instance of the modeling element type.

The result of applying the stylesheet from figure 3 to the XML document shown in figure 2 is shown in figure 4. It can be seen that the name of the class, prefixed by an M, is used as the interface name and that the <xsl:for-each>-loop has added the superclass to the extends list in the interface definition.

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:include href="interface-attribute.xsl"/>
  ...
  <xsl:template match="class">
    <xsl:variable name="basepackage">
    ...
    <xsl:call-template name="copyright"/>
    ...
    public interface M<xsl:value-of select="@name"/>

    <xsl:if test="not(@name='Base')">
      extends
      <xsl:if test="count(superclasses/classref)=0">
      MBase
      </xsl:if>
    </xsl:if>

    <xsl:for-each select="superclasses/classref[@isParent='true']">
      <xsl:variable name="class" select="document(@href)/class"/>
      <xsl:choose>
        <xsl:when test="not($class/@namespace=$actualpackage)">
          <xsl:call-template name="createFullyQualifiedClassName">
            <xsl:with-param name="basepackage" select="$basepackage"/>
            <xsl:with-param name="actualpackage" select="$class/@namespace"/>
            <xsl:with-param name="class" select="$class/@name"/>
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          M<xsl:value-of select="$class/@name"/>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:if test="not(position()=last())">,</xsl:if>
    </xsl:for-each>
  {
    ...
  }
  </xsl:template>
</xsl:stylesheet>
```

Figure 3: Example of stylesheet implementation

```
package de.unibwm.ist.mof.model;

import de.unibwm.ist.mof.*;
import de.unibwm.ist.mof.undo.*;

import java.util.Collection;
import java.util.List;


public interface MClassifier extends MGeneralizableElement {
    //attributes
    // association ends
    // resources
}
```

Figure 4: Example of generated code

4 Implementation

The implementation of MOmoC corresponds closely to the transformations steps which were described in section 2. Basically it consists of a frontend which generates the internal XML representation and a backend which controls the XSLT processing. We decided to implement the frontend in Java. This allows us to use generated code for the parser and the object representation. These parts can be generated for all MOF-based languages in order to facilitate the building of model compilers for domain-specific modeling languages. Additionally the frontend contains the generator for the internal representation which serves as interface between frontend and backend.

The MOmoC package consists of the four main subpackages, which are represented by the UML packages Parsers, Modules, Generators and Formatters in figure 5. The fifth package, MOMOC, contains the "driver program", which controls the generation process.

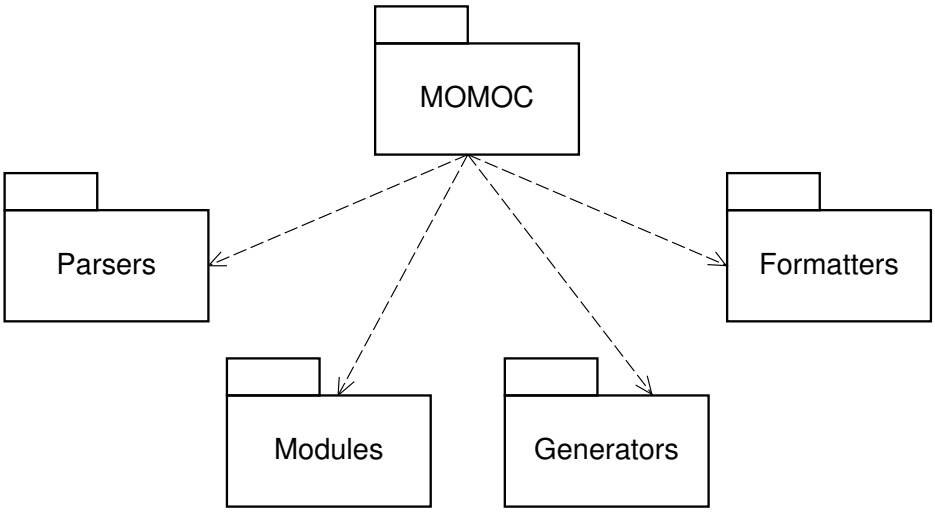

Figure 5: Architecture of the MOmoC

The Parsers package mainly contains generated code to read XMI documents and build an object representation of the MOF model. Currently, the generated code is compatible to the code of the nsuml-library ([5]), which is the basis for the ArgoUML ([2] modeling tool.

The compatibility facilitates the building of modeling tools for MOF or UML profiles using ArgoUML as basis for the user interface implementation. Therefore,the backend for nsuml-compatible libraries was developed first and the *MOMo Compiler* was initially built using the code for its internal model representation. It

is planned to switch to a generated JMI ([3]) implementation, when JMI is available for MOF 2.0.

Beside the generated parser the Parsers package contains hand-written parsers for MOF 1.4 and UML 1.5 which were used to bootstrap the compiler and are currently used to import models from UML tools as long as no MOF modeling tool is available. The handwritten parsers were originally implemented for MOF 1.4 and are now implementing the rules for transforming MOF 1.4 to MOF 2.0, which are specified in [1], chapter 11.

The object representation created by the parser can be modified by modules before the internal XML representation is generated. Currently the MOmoC implementation contains modules for resolving naming conflicts and mapping types. Naming conflicts are resolved by adding a number to the end of one of the conflicting names. The type mapping is needed to map the MOF model to an implementation technology. This is especially needed for models which contain their own datatype definitions, which need to be mapped to MOF or target language data types. The user can implement his own modules to do whatever manipulation he wants. Every module has to implement the interface Module, which is defined in the package Modules.

The Generators package contains two generators. The XML generator transforms the modified object representation to the internal representation in XML. Currently, the generator for the internal representation creates XML representation for MOF-models only. For other languages, e.g. UML, the XML generator needs adaptation. It is planned to replace the current XML generator by a generic implementation in a later version of the MOmoC.

The internal representation is used as basis for the code generation by the Code generator afterwards. The code generator applies XSLT stylesheets to the XML document to generate code in the target language. The code generator is configurable to apply a number of stylesheets to documents containing specific model elements. For example in JMI ([3]) for each class in a MOF model an interface of the class and a proxy which serves as factory for instantiating the class are generated. To achieve this generation with the MOmoC two stylesheets need to be applied to all documents which contain informations about classes. Therefore, it is possible to configure the system in a way that it searches for all documents which represent classes and apply a set of stylesheets to these documents.

After finishing the code generation process an optional code formatting process can be started. The formatters for the different target languages are located in the Formatters package. This optional step is included, because the output of the XSLT transformation is in many cases ugly formatted and therefore difficult to read and debug. Thus, the code formatters only exist to facilitate backend development.

5 Summary

In this paper we describe the *MOmo Compiler*, a flexible tool to generate implementations from meta-model definitions. In its default configuration the tool conforms to the MOF 2.0 standard, but it is extensible to support other languages as well. In order be as flexible as possible, the compilation process is done in two steps. Firstly the XMI input is transformed into an internal XML representation and secondly transformed to code by applying XSLT stylesheets.

In the first phase of the MOmo project we concentrated on the mechanisms to implement the model compiler. Our main goal for the next project phases is to increase the usability of the compiler by developing suitable interfaces for configuration and backend development. Additionally we plan to build tools for designing meta-models based on ArgoUML ([2]) or Eclipse ([4]).

References

[1] Adaptive Ltd, Ceira Technologies Inc., Compuware Corporation, Data Access Technologies Inc., DSTC, Gentleware, Hewlett-Packard, International Business Machines, IONA Technologies, MetaMatrix, Rational Software, Softeam, Sun Microssystems, Telelogic AB, Unisys, and WebGain. *Meta Object Facility (MOF) 2.0 Core Proposal*, April 2003. ad/2003-04-07.

[2] ArgoUML. http://www.argouml.org.

[3] Ravi Dirckze. *Java™Metadata Interface (JMI) Specification, Version 1.0*. Unisys, 1.0 edition, Juni 2002.

[4] Eclipse. http://www.eclipse.org.

[5] Novosoft. Novosoft UML Library (NSUML). http://nsuml.sourceforge.net.

[6] Object Management Group. *Meta Object Facility (MOF) 2.0 XMI Mapping*, April 2003. ad/2003-04-04.

[7] U2 Partners. *Unified Modeling Language: Infrastructure, Version 2.0*, März 2003. ad/2003-03-01.

[8] U2 Partners. *Unified Modeling Language: Superstructure, Version 2.0*, April 2003. ad/2003-04-01.

[9] W3C. *XSL Transformations (XSLT) Version 1.0*, November 1999. W3C Recommendation, http://www.w3.org/TR/xslt.