A flexible code generator for MOF-based modeling languages

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

August 30, 2003

Abstract

The Meta Object Facility (MOF) is the core component of OMGt's Model-Driven Architecture (MDA). MOF defines the metamodeling language, which is used to define the languages which are used to model apllications, such as the Unified Modeling Language (UML). The importance of MOF will grow in the future, because more and more domain-specific modeling languages will be defined as extensions to or adaptations of the UML instead of being proprietary.

To be really useful for software development, modeling languages need to be accompanied by tools. In this paper we present MOmoC, which is a model compiler based on the Extensible Stylesheet Language Transformations (XSLT) standard. MOmoC generates implementation code from XMI representations of MOF-metamodels. The compiler frontend mainly consists of generated code and creates an object representation from the XMI input file. This object representation is transformed into an internal XML representation, which is easier to process by XSLT than the input format XMI. The compiler backend generates implementation code by applying XSLT stylesheets to the internal XML representation of the MOF model.

This architecture facilitates the adaptation of the generated code to specific purposes as well as the adaptation of the compiler to other MOF-based modeling languages. Therefore, we think that MOmoC can serve as a basis for automating the model transformations within MDA-based processes.

1 Introduction

Model-driven software development has numerous advantages over manually developed software. The most important are: higher level of abstraction, facilitated maintainability, less erroneous code and higher re-usability ([5]). The higher level of abstraction allows the developer to concentrate on the important parts of the system and to leave the details to a compiler. This increases productivity because the developer does not need to deal with details, creates consistent implementations of designs and facilitates to deploy one design to several implementation technologies and/or platforms.

The advantages of modeling languages compared to programming languages are the same as the advantages of high-level programming languages compared to assembly languages. Therefore the availability of compilers for modeling languages

would lead to the replacement of high-level programming languages by modeling languages similar to the replacement of assembly languages by high-level programming languages.

For these reasons more and more areas of software development are currently switching from traditional to model-driven approaches. Therefore, the *Object Management Group (OMG)* ([9]) developed the *Model-Driven Architecture (MDA)* ([6], [7]) to define a standardized approach to software development based on models. Each phase within a software development process generates additional informations which need to be added to the model by a suitable transformation. These transformations should be supported by tools in order to facilitate process execution and increase the accuracy of the transformations.

In this paper we present the MOmo Compiler (MOmoC), which is a model compiler based on OMG and W3C standards. MOmoC generates implementation code from the XMI ([10]) representation of models which conform to MOF 2.0 ([1]). In the following sections we provide an overview over the architecture of MOmoC (Section 2) and show the steps which are carried out during a compilation process (Section 3). Finally we summarize the results and provide an outlook on future work within the MOmo project.

2 The MOmo Compiler

MOmoC consists of a frontend which generates the internal representation and a backend which generates the implementation code. The frontend mainly consists of generated Java classes, which implement an XMI Reader and a repository for MOF-compliant metamodels. The generated code is responsible for reading an XMI file and building an object representation of its content. The code is generated by MOmoC itself and it is possible to generate the frontend for any other MOF-compliant modeling language, which greatly reduces the costs for adapting MOmoC to these modeling languages.

The object representation created by the parser can be modified by user defined modules. Currently the MOmo Compiler 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 modified object representation is transformed to XML documents in order to be able to use *Extensible Stylesheet Language Transformations (XSLT)* ([11]) to generate implementation code. This is the only input language specific part within MOmoC and it is planned to replace this by a generic XML generator in future versions.

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. It is even possible to mitigate the performance loss compared to backends which are written in a programming language by compiling the XSLT stylesheets to Java.

An optional formatting step finalizes the compilation process. This step is included, because the output of the XSLT transformation is in many cases badly formatted and therefore difficult to read and debug. Thus, the code formatters only exist to facilitate

backend development.

On implementation level the MOmo Compiler consists of the four main building blocks, which correspond to the compilation steps we just described. The blocks are represented by the UML packages Parsers, Modules, Generators and Formatters in figure 1. The fifth package, MOMOC, contains the "driver program", which controls the generation process.

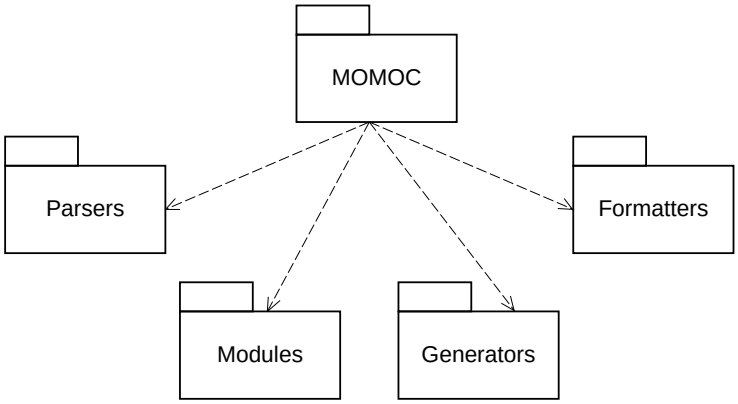

Figure 1: Architecture of the MOmo Compiler

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 ([8]), 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 user defined modules are located in the Modules package. All modules implement the predefined interface `momoc.Modules.Module` and their code is executed before the XML generation process starts. The MOmoC standard implementation contains the `DataTypeMappingModule` which allows to map the datatypes of the model to datatypes of the target language.

The Generators package contains two generators. The *XML generator* transforms the modified object representation to the internal representation in XML. The internal representation is used as basis for the code generation by the Code generator afterwards. The main difference between XMI and internal 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. This allows a more efficient code generation process, because in most cases the implemen-

tation code is generated for single model elements and not for the whole model. An example for an implementation which contains code per model element is JMI.

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.

The *code generator* applies XSLT stylesheets to the XML document to generate code in the target language. The code generator is configurable to apply any 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 is generated. To achieve this generation with the MOmo Compiler 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.

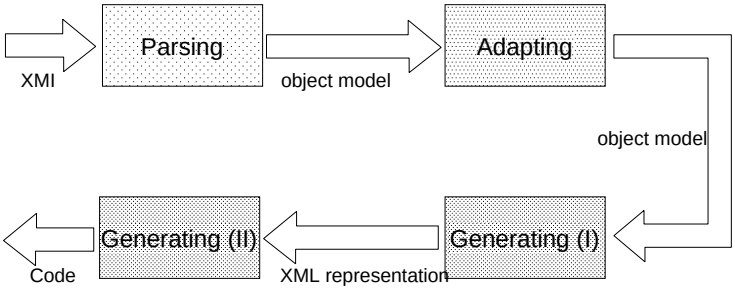

Figure 2: Compilation steps

The code formatters for the different target languages are located in the Formatters package. Currently, the package only contains formatters for XML, DTDt's and Java. Any MOmoC formatter implements the predefined `CodeFormatter` interface. Additional formatters for other target languages which also implement this interface are pluggable into MOmoC.

3 Compilation steps

In the following we show a small example from the MOF 1.4 specification, which shows the steps of the generation process carried out by the MOmo Compiler. The processing starts with an XMI-file, which contains the representation of the MOF 1.4 metamodel. We will concentrate on the CLassifier class within the MOF 1.4 metamodel to show the results of the compilation steps.

Figure 3 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 sub-nodes 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

```
<?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 3: Internal representation of MOF 1.4 meta-class Classifier in XML

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 code in the target language by applying stylesheets. Figure 4 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.

```
<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 4: Example of stylesheet implementation

The result of applying the stylesheet from figure 4 to the XML document shown in figure 3 is shown in figure 5. 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.

```
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 5: Example of generated code

4 Summary

This paper describes 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 flexible the compilation process is done in four steps.

The first step, generating an object representation from the XMI input file is implemented by generated code. This code can be generated for any MOF 2.0 compliant metamodel which makes MOmoC easiliy adaptable to other modeling languages. In a second step the object representation can be modified by user defined modules in order to implement specific tasks such as mapping model datatypes to datatypes of a target programming language.

Within the third step an interal XML representation is created from the object representation. Processing these XML representation is easier and more effective than processing the original XMI representation, because it consists of several small files which represent single model elements instead of one large representing the whole model. The final step within the compilation process generates code from the XML representation by applying XSLT stylesheets.

Our experiences with using XSLT as a template language for code generation are ambivalent. The advantages in regard to other template languages are better documentation and tool support. The main disadvantage is the verbosity of XSLT which tends to lead to large and complex stylesheets. Additionally, the current version 1.0 of XSLT is missing some text manipulation features like which are often used while generated code. Currently we add these missing features by callbacks to Java code and version 2.0 of XSLT will include the necessary functions.

In the future we will work on the following topics:

- Support for the design of MOF models based on ArgoUML ([2]) and/or Eclipse ([4])
- Increase the usability of MOmoC by providing user interfaces for configuration and backend development

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^{TM}Metadata Interface (JMI) Specification, Version 1.0*. Unisys, 1.0 edition, June 2002.

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

[5] Thomas Kühne. Automatisierte Softwareentwicklung mit Modellcompilern. *thema Forschung*, pages 116–122, January 2003. (in german).

[6] Joaquin Miller and Jishnu Mukerji. *Model Driven Architecture*. Object Management Group, July 2001. Document number ormsc/2001-07-01.

[7] Joaquin Miller and Jishnu Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, June 2003. Document number omg/2003-06-01.

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

[9] Object Management Group. http://www.omg.org.

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

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