The Bosco Project
A JMI-Compliant Template-based Code Generator

Pascal André Gilles Ardourel Gerson Sunyé
LINA, University of Nantes
2, rue de la Houssinière – BP 92208
44322 Nantes Cedex 03 – FRANCE
[Pascal.Andre,Gilles.Ardourel,Gerson.Sunye]@lina.univ-nantes.fr

Abstract

Bosco is a code generation tool, which goal is to accept any MOF model as parameter and follow the evolution of OMG standards. It generates the underlying model (also called repository) for any modeling language expressed in MOF. In other words, it reads XMI files and generates the corresponding source code, in different object-oriented languages (Java, Eiffel, Python, C++). In the case of Java, the generated code implements the JMI specification. In Bosco, any user can program add-ins at each level using the visitor pattern and the template technology.

Keywords: Software Engineering, MOF, UML, JMI, Templates, Model Transformation.

1 Introduction

Since its first version the UML – as well as other related OMG standards such as the Meta Object Facility (MOF), the XML Metadata Exchange (XMI) or the Object Constraint Language (OCL) – underwent several versions, making it very difficult to a modeling tool not only to be kept up to date, but also to manage models expressed in different UML versions. The most common solution to keep a tool up to date is to generate automatically the code used to represent models, called the underlying model (or repository), which is highly dependent from the UML specification. This is the same approach as the one used by meta-tools [4]: use the specification of a modeling language to generate its underlying model.

Concerning the UML, its specification (or at least its syntax) is expressed by a model, also called metamodel, which is represented in MOF [6] and proposed as a XMI document [**?**]. The MOF is a subset of UML, roughly its class diagram, while XMI is a XML document, used to exchange models. However, MOF and XMI are also frequently updated and the particularly high levels of abstraction handled by this type of tool, as well as the unusual merge of code and strings containing code, have both deleterious effect on its maintainability. Furthermore, since the generated code does not follow any standard, changing from a generation tool to another is almost impossible. Therefore, generation tools face exactly the same difficulties as UML tools, but on a lesser scale.

The generative approach is used for instance in ArgoUML [1], which uses a tool called NSUML [8] to generate its underlying model and which faces nowadays an evolution dilemma. Since NSUML is not evolving anymore, Argo is undergoing an important development effort to move to another underlying model and support the recent versions of the UML. The goal of the Bosco project is to provide underlying models to modeling tools. It attempts to solve the difficulties delineated above thanks to three main properties.

1. The generated code respects a common specification for metadata interfaces (JMI), meaning that the tool that uses this code is not dependent from Bosco.
2. Thanks to the use of templates (see § 3), the readability of its implementation code is ameliorated.
3. As Bosco underlying model is itself generated, Bosco is ready to deal with the evolution of standards.

The paper is organized as follows. After a short presentation of Bosco in section 2, its architecture is described and illustrated by an example in section 3. Section 4 focus on the generated environment. We compare Bosco to other approaches in section 5 before concluding.

2 Scope

The Bosco project[1] is a generic open source project that focus on the internal representation of sofware development models. Bosco parameters are XMI files describing metamodels and meta-metamodels. It currently works for MOF release 1.4 and UML releases. Its entries are software development models (e.g. UML models, ER models, ...). Bosco add-ins are model driven functions: the user implements modeling functions such that model checking, code generation, documentation generation, model transformation (e.g. from UML 1.5 to UML 2.0) and so on.

Bosco is standard compliant for inputs and outputs parameters (OMG standards : MOF, UML, XMI). Currently the generated code implements the JMI specification [3]. Bosco handles the three higher levels of the OMG standard [7] (infrastructure p. 31).

1. Meta-metamodel : Bosco is assumed to read MOF specifications (`M3`) for building the metamodel compilers. It currently accepts MOF 1.4.

2. Metamodel : Bosco reads metamodels (`M2`) for building model compilers. The tool has been tested with UML 1.4 and UML 1.5.

3. Model : Bosco loads `M1` models to apply user-defined operations such as model checking, code-generation, testing, metric computations, etc.

The instance level is not taken into account since no execution of UML models is provided.

3 Bosco Architecture

Bosco has a two part architecture, reflecting the two conceptual levels it handles. The first part (Figure 1) handles language specifications (or metamodels): it reads XMI-MOF documents and generates their underlying models. The second part (Figure 2) is the generated environment: it reads and writes XMI documents that are compatible with the language specified in the first part. In the current release, this part is implemented (i.e. generated) in Java and is compatible with the Java Metadata Interfaces (JMI).

3.1 Implementation

The specification-level part of Bosco is written in Python [11] and consists of a MOF underlying model, a Sax-based XMI parser and a code generator. The generator uses a template engine named Cheetah [9] and is driven by template files, which merge source code with basic control structures (`#for`, `#if`, etc.). Thanks to this, the generation is highly configurable: one can add new features to the generated code without writing a single line of Python.

Our goal when implementing this part was to make it as simple as possible: the input specifications are supposed to be valid and the MOF underlying model is minimal. However, the templates that will generate a more robust parser and a full MOF underlying model are under construction.

One may accurately wonder why this part of Bosco was not implemented in Java, since its intent is to generate Java code and some parts of it (e.g. the XMI parser) could be shared between the two architectural parts. Three main reasons motivate this choice.

The first one, and maybe the most important, is that Cheetah uses Python objects as the context for template expansions, allowing the template to access its attributes and methods. Thus, all Python objects are potentially accessible, while in the available Java template engines, such as Velocity[2] or WebMacro[3], only the values of a dictionary (filled before the expansion) are accessible. This means that when a template needs to access objects that are not in the dictionary, the code calling that template has to be replaced.

The second reason is that lists, as well as list operations[4], are smoothly handled in Python. Since a model can be seen as a succession of lists (e.g. a package contains a list of classes, a class contains a list of attributes and so forth), Python is particularly adapted to handle models. For instance, when creating the string representing the parameter list of a MOF class constructor, one must find a list of all its instance-level single-valued attributes and then translate this list into a string containing their type names followed by their names, separated by commas. While in Python this manipulation can be achieved by a sequence of 3 operations (filter, map, join) in a single line of code, it would need several lines in Java.

The third and last reason is that we wanted the specification level to be independent from both the generated environment and its implementation language, and a simple manner to accomplish this is to use a different language.

[1] http://bosco.tigris.org

[2] http://jakarta.apache.org/velocity/index.html

[3] http://www.webmacro.org

[4] map, filter and reduce

Figure 1 : *architecture*

3.2 Code Generation Example

In this section, the code generation process is illustrated by a small example. Bosco reads XMI-MOF specifications and instantiates its own implementation of the MOF underlying model. This first step is necessary in order to translate the tree structure of a XML document into a graph, which is more adapted to code generation. Once a MOF model is instantiated, it is used as the context for the templates.

XMI Element The element below comes from the UML 1.5 specification. It describes the metaclass `Class`, and its single attribute, `isActive`:

```
<Model:Class annotation="" isAbstract="false"
   isLeaf="false" isRoot="false" isSingleton="false"
   name="Class" supertypes="a32989FB2023D"
   visibility="public_vis" xmi.id="a3298A02900FE">
   <Model:Namespace.contents>
     <Model:Attribute annotation="" isChangeable="true"
        isDerived="false" name="isActive"
        scope="instance_level" type="a33DD6F650276"
        visibility="public_vis" xmi.id="a33F24B5A0190">
        <Model:StructuralFeature.multiplicity>
           <Model:MultiplicityType is_ordered="false"
           is_unique="false" lower="1" upper="1"/>
        </Model:StructuralFeature.multiplicity>
     </Model:Attribute>
   </Model:Namespace.contents>
</Model:Class>
```

Templates A Cheetah template file merges raw text with Python code and some simple statements (`#set`, `#if #else`, `#for`). These statements can directly reference Python objects, which work as parameters for template expansion. An example of a template is presented below. It is a simplified version of the one that generates Java interfaces for MOF classes. An instance of a MOF Class (the context) is accessible through the *$context* variable.
When the template is expanded, Cheetah proceeds as follows. First, a string containing the names of the context super-classes, separated by commas, is created. Then, variables are replaced by their content: the qualified name of the package containing the context and the name of the context. Finally, Cheetah will create two methods for each attribute of the context class, using their names and types as parameters.

```
#set $parents_str = ', '.join(map(lambda x:
        x.java_full_name, $context.supertypes))
package $(context.container.java_full_name);
public interface $(context.java_name)
        extends $parents_str {
        /**          Attribute accessors          */
#for attrs in $context.attributes()
        public void $(attrs.asSetter)($attrs.type.
              java_full_name $(attrs.asArgument))
                 throws javax.jmi.reflect.JmiException;
        public $(attrs.type.java_full_name)
                $(attrs.asGetter)()
                 throws javax.jmi.reflect.JmiException;
#end for
}
```

Figure 2 : *Generated environment*

Generated Java code An example of the above template expansion, using the MOF class *UmlClass* as context, is presented below. Since this class owns a single attribute, only two methods are created. It is important to notice that all the text other than the Cheetah statements is copied as is, meaning that the generated code is quite readable.

```
package org.omg.uml.foundation.core;
public interface UmlClass
    extends org.omg.uml.foundation.core.Classifier {
    /**      Attribute accessors      */
    public void setIsActive(java.lang.Boolean a_isActive)
            throws javax.jmi.reflect.JmiException;
    public java.lang.Boolean isActive()
            throws javax.jmi.reflect.JmiException;}
```

4 The generated environment

For each input metamodel, Bosco generates a Java environment composed of JMI compliant classes, visitors and a XML description used by model management classes.

4.1 Java Metadata Interface

JMI is a platform-independent specification for manipulating metadata using MOF, XML and Java. JMI is composed of a set of interfaces and a set of code generation rules that guide the implementation of MOF specifications in Java. The JMI interfaces stipulate a common behavior for any MOF specification, while the generation rules define how language-specific behavior should be implemented. For instance, the metaclass *UmlClass* presented in 3.2 should implement two analog methods allowing to read the attribute `isChangeable`. The first one, `refGetValue(String)`, is generic: it takes the name of an attribute as a parameter and returns its value. The second one, `isChangeable()` is specific: it only returns the value of the attribute.

The JMI specification involves several MOF concepts (classes, attributes, packages, associations, constraints, operations, data types and references) and is somewhat complex. Consequently, only the part of it that concerns the implementation of MOF classes will be explained here. To make it simple, a MOF class is implemented by two Java classes. The first one is its underlying representation which implements its attributes, operations and references. The second one is called a *proxy*: it acts similarly to Smalltalk metaclasses: its role is to create instances of the former and keep a list of them. A similar principle is used for MOF associations.

4.2 Visitors

While one certainly can use the JMI-compliant code that represents a metamodel `M2` as is, several facilities are provided to help tool developers.

First, Bosco generates for each language specification `M2` a *XMIHandler* that can read a XMI file containing a model `M1` and load its representation as the instances of the generated JMI-compliant classes.

Second, sample visitors are generated for each metamodel and provide a simple browsing of the runtime representation (i.e the instances of `M2`) of the loaded models `M1`.

Creating visitors

The expressiveness obtained by only using elements from the MOF underlying model is often sufficient to describe how objects are being traversed by visitors.

However, most of the functionality to be provided by visitors is tied to a specific metamodel `M2`. The creation of Bosco add-ins is done easily by extending the visitors provided in Bosco. This can be done in multiple ways:

- creating a Java subclass of an existing visitor and implementing only the visit methods needed
- creating a Cheetah template that will generate a Java subclass of an existing visitor

The following lines are sufficient to describe the generic *visit* method which displays the type of entity for each instance of a loaded model.

```
public class SimpleVisitorConsole
    extends AbstractVisitor {
// For All concrete classes,
#for $element in $model.all_contained_classes()

public void visit$(element.name)
   ($(element.java_full_name) element){
   System.out.println
       ("Visit Entity of Type : $(element.name)");
// in a UML Model, the element has a Name :
       System.out.println(((ModelElement)element).
          getName());}
          super.visit$(element.name);
}// end visit$(element.name)
#end for
}
```

Common parts of the `visit` methods can be factorized in a Visitor template (like the `SimpleVisitorConsole`), then differences can be specified in a Java subclass (for linking to an external program in the following example).

```
public class UMLVisitor
    extends AbstractVisitor {
public void visitClass(
  org.omg.uml.foundation.core.UmlClass element){
    myboscoaddins.ExternalApp.manageClass(element);
}
```

4.3 Easy model manipulation

Bosco produces a XMI file that lists the metamodels generated and the associated handler and visitor classes.

Manipulating concurrently models from different metamodels is possible with the `MOFRepository`, `ModelRepository` and `ModelM1` classes, that handle sets of metamodels, specific metamodels and models, respectively. These classes propose loading and visiting facilities.

The following code searches for the UML1.5 metamodel JMI representation, loads a XMI model and visits it.

```
ConfigRepository metahome=new ConfigRepository
       ("/home/bosco/config/gen_sources.xml");

 ModelRepository umlmodelM2=metahome.getModel("uml");
 ModelM1 m1=umlmodelM2.loadInstanceM1
       ("/home/bosco/examples/test.xml");

// visiting using the gen_sources.xml info
 umlmodelM2.visitModelWithName(m1,
              "SimpleVisitorConsole");

// visiting using a visitor directly
 umlmodelM2.visitModelWithObject(m1,
                new myboscoaddins.UMLVisitor());
```

5 Related work

Generating an underlying model from a language abstract syntax is not a new technique. It is called *meta-modeling* and had been used on several academic and commercial tools, called *meta-tools*, since the decade of 80. The first tool that applied this technique to the OMG standards is probably NSUML [8], from Novosoft, which has been used in ArgoUML and in the first releases of Poseidon. The present version of NSUML implement the JMI and is a direct concurrent to MDR [5] from Sun, CIM [10] from Unysis or ModFact [2].

All these tools differ from each other on some features that are included in the generated underlying model: events, transaction, persistence, etc. The code generation techniques used by these tools are quite similar: the code generator is written in java, as a succession of prints.

Bosco has a different goal: it does not focus on the quantity of services generated, but on the process of generation itself. It does not intend to generate a more complete environment than MDR, for instance, but to propose a more flexible generation process. The flexibility is reached thanks to Cheetah templates, which increase the code readability by separating the generator behavior from the code that will be generated. In other words, templates eliminate the *noise* that masks the generated code: string concatenation, indentation calculation, space insertion, etc. Thus, the

knowledge needed to understand and modify a template is reduced to MOF and the few Cheetah control structures.

6 Conclusion and future work

Implementing a code generation tool is an interesting task that helped us to validate our technological choices and to observe some difficulties. More precisely, the choice of Python and Cheetah as generation engine, as well as Java and JMI as the first implementation target proved to be correct. The MOF underlying model and the XMI parser became promptly stable, while the complexity of the JMI showed that all the needed data was accessible from the templates. When developing the templates, the problems we encountered concerned the implementation of the required JMI behavior and not the retrieving of data from the MOF model.

The main difficulties we had came from an unexpected source, the OMG specifications. The different specifications of the UML, for instance, are expressed in different versions of MOF and are written in different versions of XMI, making the goal of implementing an *universal* tool that could read any MOF specification a very difficult task. This goal will only be reached when the specification-level part of Bosco will be partially generated, allowing different versions of MOF to be handled.

The goal of making Bosco an open tool, allowing users to easily configure the code generation will only be demonstrated when users, others than Bosco developers, will start to write new templates. While the present templates seem to be very readable, we still think we can improve their readability and extensibility, using some unexplored Cheetah functionalities, such as template inheritance and functions.

The code generated by the current release of Bosco implements most of the JMI specification. It lacks, however, a complete support for the metamodel (the language specification), which should be accessible from the model level. The full JMI implementation will be available in the following months. The next release of Bosco will include a OCL parser, which is under completion, and several facilities for code generation. The OCL parser will be combined with an evaluator, allowing Bosco to verify if the well-formedness rules of a given language are respected by any model expressed in this language.

References

[1] ArgoUML. ArgoUML - a modelling tool for design using UML, 2004. http://argouml.tigris.org/.

[2] Xavier Blanc, M-P. Gervais, and R. Le Delliou. The specifications exchange service of an RM-ODP framework. In *4th International Enterprise Distributing Object Computing Conference (EDOC'00)*, Germany, sept 2000. IEEE Press.

[3] Ravi Dirckze. Java metadata interface (JMI) specification. Technical Report JSR 040, Unisys Corporation and Sun Microsystems, http://java.sun.com/products/jmi/, June 2002.

[4] Steven Kelly, Kalle Lyytinen, and Matti Rossi:. Metaedit+: A fully configurable multi-user and multi-tool CASE and CAME environment. In *CAiSE*, pages 1–21, 1996.

[5] Martin Matula. Netbeans metadata repository. Technical report, Sun Microsystems, 2003.

[6] OMG. Meta Object Facility(MOF) Specification Version 1.4. Technical report, Object Management Group, http://www.omg.org, April 2002.

[7] OMG. The Unified Modeling Language Specification, version 2.0 rfp. Technical report, Object Management Group, http://www.omg.org/, 2004.

[8] Constantine Plotnikov. Novosoft metadata framework. Available at http://nsuml.sourceforge.net/.

[9] Tavis Rudd, Mike Orr, and Ian Bicking. Cheetah: The python-powered template engine. In *The Tenth International Python Conference*, Alexandria, Virginia, February 2002.

[10] CIM Development Team. JMI-RI documentation. Technical report, Unisys Corporation, October 2002. Available at http://ecommunity.unisys.com/.

[11] Guido van Rossum. Python reference manual. Technical Report CS-R9525, CWI, May 1995.