Model-driven generative approach for concrete syntax composition

Pierre-Alain Muller
pa.muller@uha.fr
UHA/INRIA – France

Philippe Studer
ph.studer@uha.fr
ESSAIM/MIPS – Université de Haute-Alsace - France

Jean-Marc Jézéquel
jean-marc.jezequel@irisa.fr
Irisa/Université de Rennes – France

Abstract. This position paper presents a model-driven generative approach for composing concrete syntax. Concrete syntax is generated from information contained in a business model (represented with UML class diagrams) and from textual templates (represented in a composition model). The approach was originally implemented in the context of Web engineering (generation of HTML text) and was latter applied to paper catalogue generation (generation of QuarkXpress textual serialization format).

Our contribution to the workshop would be to discuss how our approach could be applied to model generative programming, to gather feedback and to foster the discussion.

1 Introduction

In the context of the Netsilon[1] project, a model-driven tool for automatic generation of Web information systems, we have designed and implemented a hypertext model and associated model transformations to generate programs (in Java or PHP) so as to generate HTML text on the fly, at execution time of the Web application.

Latter on, we have applied the same technique to generate multi-channel (electronic and paper) catalogues for a tour operator. This multi-channel generation used exactly the same modelling elements (defined in a hypertext meta-model), the details of the channels were captured by specific models conforming to the hypertext meta-model.

More recently, we realized that a subset of the hypertext meta-model (the composition meta-model) could probably be used as well as a meta-model for generative programming.

In this paper, we show how we generate Java programs (Servlet and JSP) or PHP programs, which in turn generate concrete syntax (such as programs or documentation).

The following picture shows the production tool chain that we use. The models of the business (expressed in a simplified UML along with an action language named Xion) and of composition (expressed with our composition meta-model) are fed in a model transformation built in Netsilon, which generates scripts (either in PHP, or in Java). In turn, when executed, these scripts extract information from the instances of the business model, read the templates, and generate the concrete syntax.

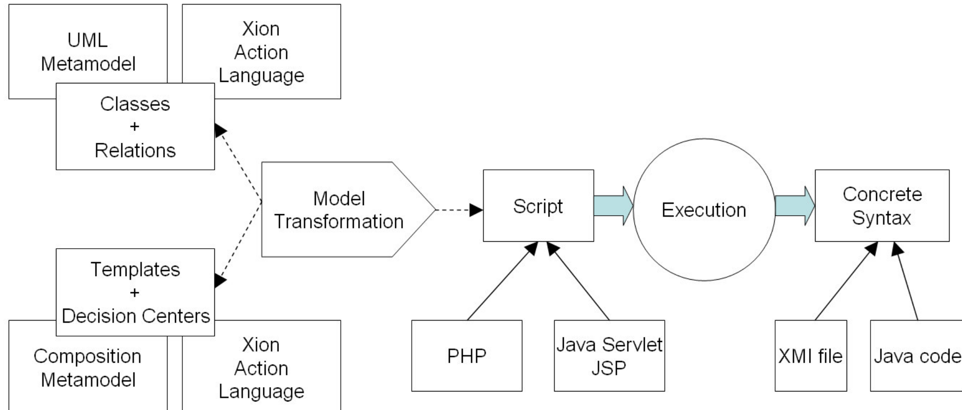

Figure 1 : Overview of the generation of concrete syntax, via our model-driven generative tool-chain.

Based on this industrial experience and on several years of research into model transformations[2], we now present our position on Model-Driven generative approach.

2 Lessons learned

2.1 Multi meta-models approach

We have experienced that several meta-models/DSL are required to cater for the various needs of generative programming (meta-model for business models and general-purpose action language providing business level operations; specific meta-models for navigation and template-composition, simplicity and power of mark-up languages to specify the layout of templates). Each time it was possible we used standard meta-models or languages (UML for business models, HTML for page layout), but it is now clear that we should be able to develop and use new metamodels each time we need it (e.g.; Xion instead of the UML Action Language for describing business level actions). Most notably, we have defined novel modelling elements (the decision centers) to represent the composition of text from business information and templates.

These several meta-models can be considered as ways of specifying various aspects[3] of the system under development. The consistency of the aspects is ensured by the compilation of the Xion statements which are either used to implement the business methods, or to specify the constraints and expressions in the composition models. Final aspect weaving is performed at execution time, when the generated code is executed.

2.2 The action language can be used to write model-transformations

We have defined a multi-purpose action language named Xion. The concrete syntax of the control-structures of Xion is very close to Java, and developers felt very comfortable with that. The style of writing remains imperative, with the additional power of traversal and query of OCL upon which Xion is based.

At the model level, Xion can be used to specify the methods and to express the actions in the composition model. During code generation, the Xion statements are translated into executable code.

Interestingly, at the meta-model level (which simply means to load a meta-model into Netsilon, and then to define the models via the automatically-generated object administrator), Xion can be used to express model transformations. As Xion can both read and write instances (of the meta-model elements) it can be used to transform constructs from one model into another one.

2.3 Template-based generation

Xion alone could be used to generate text from the models, using its text input-output capabilities, but this would be cumbersome, and therefore we have added a template-based generation mechanism to serialize a model into a concrete syntax.

We have associated templates and models via the composition meta-model. Templates specify the layout while composition models express how templates are decorated with business information and composed. Templates

are read dynamically at execution time of the generated programs, so that the final layout (and content) of the templates can be modified without re-compiling the model.

This way, end-users keep a high level of control over the final generated text, without having to master the models and model-driven tool-chain. This was of paramount importance, because it allowed deploying customizable model-driven applications, which could be fine-tuned by the graphic-designers, using their usual tools (e.g. QuarkXpress in the pre-press domain, or DreamWeaver in the Web domain).

3 Discussion

We would like to examine and discuss the applicability of our approach to generative programming. When should/could this approach be used, what kind of features are missing, does it apply only to some restricted use-cases or can it be generalized to any kind of generative programming scheme?

We have made tradeoffs between visual and textual representations, while maintaining a strong separation between the various kinds of representations. For instance, in the templates we use special placeholders to refer to the decision centers declared via the graphical notation. In some cases, like the insertion of a simple value, this approach may seem a quite heavyweight (some users have asked for a way to directly embed Xion Statements in the HTML text). Is such a strong separation desirable or is it better to mix various textual languages in the same file?

When is such model-driven approach justified, can we identify a breakeven point, both in terms of kind of applications, or in terms of development approaches? Developing new meta-models and associated tools has a cost that probably needs to be shared among multiple products in a same family. Hence we see its interest in clear connection with the product line engineering domain[4].

4 Conclusion

In this paper we have presented a meta-model for template composition, which is a subset of the hypertext meta-model originally designed for Web information system modelling with the model-driven tool Netsilon.

We have shown how this meta-model can be used to represent a template expansion process realized at runtime by programs generated from models, and how this process can be used to generate concrete syntax from models.

We believe that this meta-model could be used in the context of model-driven development combined with generative programming.

Appendixes

The composition meta-model

The composition meta-model is an abstract description of a document generated as the result of the composition of textual templates, decorated with information coming from a business model and/or execution context (current date and time for instance).

Xion (see below) is used as a query language to extract information from the business model and as a constraint language to express the various rules, which govern template's composition.

Composition includes the specification of the parameters that will be transferred from template to template, and which are used at runtime either to decide which template to select or to drive the business information extraction.

The composition meta-model makes it possible for a tool to check the coherence and the correctness of the composition of templates at model compilation time. This removes all the troubles related to parameter passing and implementation language boundary crossing (mix of interpreted languages, un-interpreted strings, parameter passing conventions…) encountered when programming generative applications by hand.

The meta-model defines the concept of decision centers, which model a decision to be taken at runtime by the program which is generating the final document. We use the following three decision centers for document (concrete syntax) generation.

Icon	Name	Description
	Composer	Composers compose templates into other templates. A composer selects a target template to be inserted in place of the placeholder
	Value displayer	Value Displayers display single values. A value displayer evaluates a Xion expression, converts the result in a character string and inserts this string in place of the placeholder in the generated text.
	Collection displayer	Collection displayers display collections of values. A collection displayer acts as a composer applied iteratively to all the items in the collection denoted by a Xion expression. For each element a specific target template may be chosen.

Figure 2 : Definition of the Decision Centers and graphic representations.

Modeling the composition of a document includes visual and textual representations.

The visual notation shows the overall composition of a document from templates, under the shape of an oriented graph which grows from the left to the right, and from the top to the bottom. The left-top-most item is the root and will be translated into an executable program which will compose the templates as indicated in the model. Parameters flow in the graph from the left to the right, and at execution-time values are passed to the template expansion process and the placeholders are filled with the actual data.

The figure below shows an example of the visual notation. Note that the .html extension for the templates means that HTML tags may be used to specify the layout of the templates.

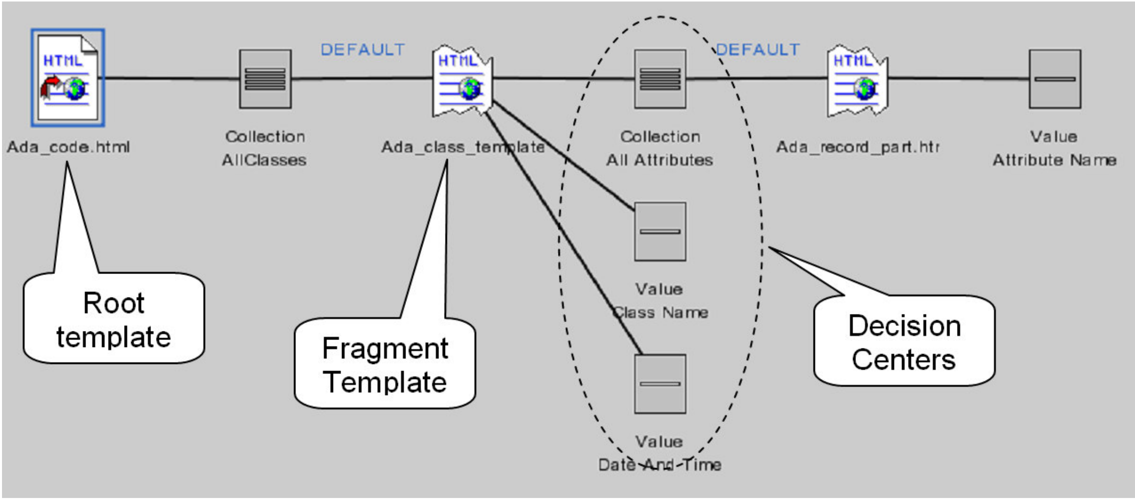

Figure 3 : Example of the visual notation for the composition of documents from templates.

In addition to this visual notation, two kinds of textual representations are used to make the models amenable to automatic translation to executable code; the Xion language and the HTML markup language.

The Xion Language

The Xion action language is used to specify the various actions to perform over the business model. Xion is based on OCL and can be considered as a concrete syntax for a subset of the Action Semantics. Xion can be considered as a precursor to recent industrial approaches such as Xactium (http://albini.xactium.com/content/) and other. It helped us build experience on what should be available in a meta-modeling tool, including the importance of an easy way to represent a composition model.

Xion can be used to:

- Create and delete an object,
- Change an attribute value,
- Create and delete links,

- Change a variable value,
- Call non-query operations.

Since most developers are already familiar with the Java language, we re-used part of its concrete syntax. Constructs we took from Java are:

- Instruction blocks, i.e. sequences of expressions,
- Control flow (if, while, do, for),
- Return statement for exiting an operation possibly sending a value,
- “super” initializer for constructors.

Moreover, for Xion to look like Java as much as possible we decided to keep Java variable declaration, and operators (==, !=, +=, >>, ? ternary operator, etc.) rather than those defined by OCL. The standard OCL library was also slightly extended, by adding the Double, Float, Long, Int, Short and Byte primitive types, whose size is clearly defined unlike the OCL Integer or Real. We have also added the Date and Time predefined types.

Xion is a multi-purpose action language. At the model level, Xion can be used to specify the methods and to express the actions in the Hypertext model. Duting code generation, the Xion statements are translated into executable code.

At the meta-model level, Xion can be used to express model transformations. As Xion can both read and write instances (of the meta-model elements) it can be used to transform constructs from one model into another one.

Xion alone could be used to generate text from the models as well, using its text-input -output capabilities, but this would be cumbersome, and therefore we have added a template-based generation mechanism, described below, to serialize a model into a concrete syntax.

Using HTML markup language to specify templates

A document is composed of templates, whose layout can be specified with HTML tags, supplemented with some special placeholders to denote the location of application of a decision center. The following picture shows how a placeholder is replaced by the content of a template at runtime.

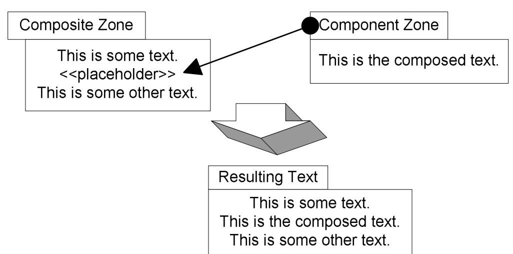

Figure 4 : At runtime, the placeholder is replaced by the component text.

The following picture shows the actual text of a template file used to generate an Ada package specification.

```
-- Generated the !-!/objexion/122 DateAndTime/<br>
--<br>
Package !-!/objexion/119 ClassName/ is<br>
<blockquote>
type Object is private;
</blockquote>
private<br>
<blockquote>
type Object_Implementation;<br>
type Object is access Object_Implementation<br>
</blockquote>
end !-!/objexion/119 ClassName/;<br>
<br>
<br>
```

Figure 5 : Example of template file.

The static text of the template is delimited and formatted by HTML tags. Dynamic text can be generated by decision centers, at locations materialized in the template by placeholders which refer to them. The syntax of a placeholder is as follows:

```
"!-!/objexion/"Decision Center Number" "Decision Center Name"/"
```

Note: the name objexion is an arbitrary chosen name, it relates to the context in which we developed the Netsilon tool.

The next section is a case study, which shows two examples of text generation from a model; XMI serialization and Java Code generation.

Case study

As a case study we will show how to generate Java code and XMI serialization from a small UML model.

The following class diagram shows the very simplified UML meta-model that we will use for the business model.

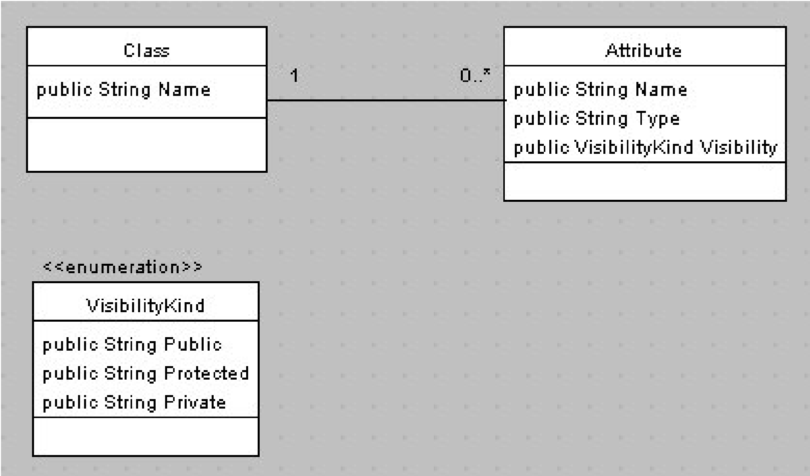

Figure 6 : Simplified UML meta-model used in the case study.

We shall create two instances of the `Class` class (`Person` and `Car`) as shown in the following class diagram.

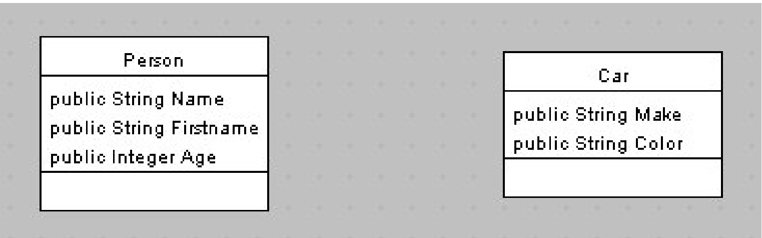

Figure 7 : Simple model, featuring persons and cars.

XMI generation

In this first part, we will show how to generate XMI serialization for a model which conforms to the simplified UML meta-model given beyond.

The following picture presents the XMI file which is generated for the simple model introduced earlier. We have used the capabilities of Internet Explorer to visualize XML files to browse through the generated XMI file. The − signs in front of the lines indicate that the lines can be elided.

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <XMI xmi.version="1.2" xmlns:UML="org.omg.xmi.namespace.UML" timestamp="2004-07-31 at 22:06:59">
  - <XMI.header>
    - <XMI.documentation>
        <XMI.exporter>Model-Driven XMI Writer</XMI.exporter>
        <XMI.exporterVersion>1.0</XMI.exporterVersion>
      </XMI.documentation>
    </XMI.header>
  - <XMI.content>
    - <UML:Model xmi.id="10jbd885657efb240434223c02b4d34e" name="GPCE workshop exemple">
      - <UML:Namespace.ownedElement>
        - <UML:Class xmi.id="10l4966db953bb9998b87c1f2c61bec1" name="Person">
          - <UML:Classifier.feature>
              <UML:Attribute xmi.id="10jaf885711f7b240434223c02b4d34e" name="Name" visibility="Public" />
              <UML:Attribute xmi.id="10j7866cdfb3a7433a6e03c06f8ccc44" name="Firstname" visibility="Public" />
              <UML:Attribute xmi.id="10jf801891921ae630425446eb40f59f" name="Age" visibility="Private" />
            </UML:Classifier.feature>
          </UML:Class>
        - <UML:Class xmi.id="10l75936378cbb5b72ae00e8a6872ff0" name="Car">
          - <UML:Classifier.feature>
              <UML:Attribute xmi.id="10j6dfa9bcf52d6b32383ca964613a3f" name="Make" visibility="Private" />
              <UML:Attribute xmi.id="10j555f34baff3c79850e8c9d962998f" name="Color" visibility="Private" />
            </UML:Classifier.feature>
          </UML:Class>
        </UML:Namespace.ownedElement>
      </UML:Model>
    </XMI.content>
  </XMI>
```


Figure 8 : Screen copy showing the visualization of the generated XMI file in Internet Explorer.

This XMI file was generated by the execution of a program which was itself generated by a model. The following picture first shows a satellite view of this model, while the next paragraphs will focus on each part in detail.

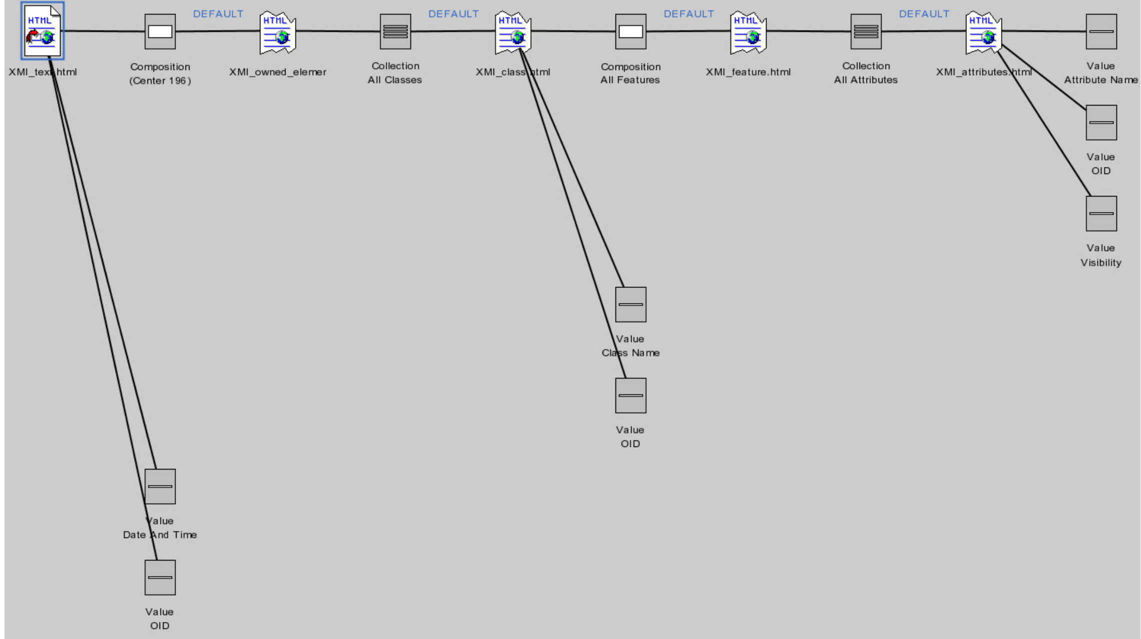

Figure 9 : Satellite view of the composition model which decribes the XMI serialization for the simplified UML meta-model of our case study.

The XMI file starts with version information and time stamp. Next, a header specifies the XMI writer which was used to generate the file. Finally, a content part contains the model elements defined in the model. Notice the representation of the two classes, and more specifically the visibility of the attributes.

Now let's have a look at the model which was used to generate the programs which in turn generated the XMI file.

The envelope part of the file was generated by the following composition model.

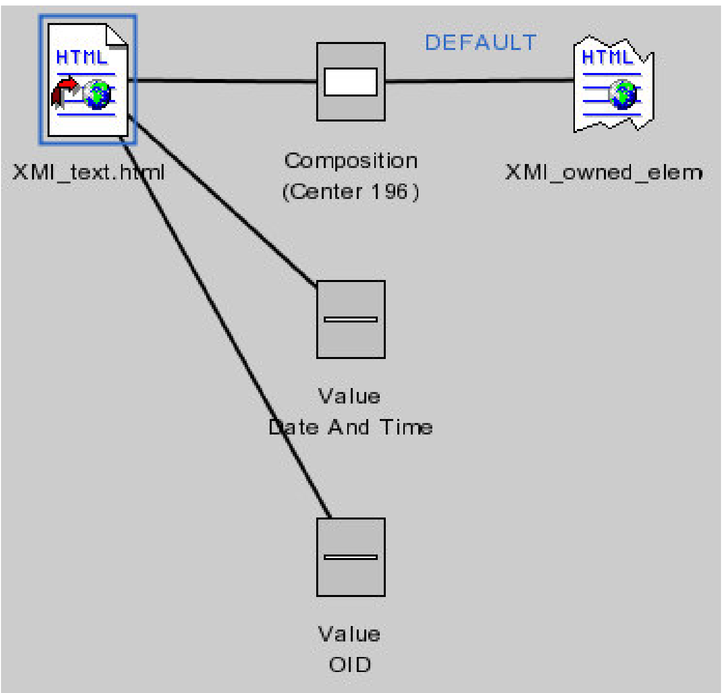

Figure 10 : Excerpt of the composition model for the envelope of the XMI file.

In this model we see that the envelope of the XMI file is made of a template named `XMI_text` and that this template references three decision centers, to embed a template named `XMI_owned_element`, and to insert two values, the date and time, and an object identifier.

The following picture shows the text of the template `XMI_text`, as well as the placeholders (denoted by their identifier and name) which refer to the three previously described decision centers.

```
<?xml version="1.0" encoding="UTF-8" ?>
<XMI xmi.version="1.2" xmlns:UML="org.omg.xmi.namespace.UML"
     timestamp="!-!/objexion/126 DateAndTime/">
    <XMI.header>
        <XMI.documentation>
        <XMI.exporter>Model-Driven XMI Writer</XMI.exporter>
        <XMI.exporterVersion>1.0</XMI.exporterVersion>
        </XMI.documentation>
    </XMI.header>
    <XMI.content>
        <UML:Model xmi.id="!-!/objexion/205 OID/" name="GPCE workshop exemple">
            !-!/objexion/196/
        </UML:Model>
    </XMI.content>
</XMI>
```

Figure 11 : Template for the generation of the envelope of the XMI file.

The content part is in turn described by the following model. The `XMI_owned_element` template contains a collection displayer decision center (`All Classes`) which will repeatedly embed the `XMI_class` template for all the classes defined in the model.

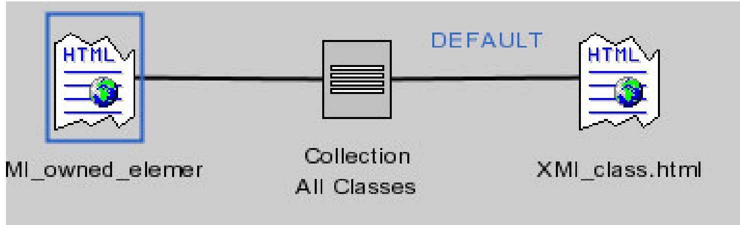

Figure 12 : The XMI_class template is expanded for all the classes contained by the model.

The `XMI_owned_element` and `XMI_class` templates are connected by a decision center of type collection displayer. This decision center models the fact that at runtime the `XMI_class` template will be inline-expanded in the `XMI_owned_element` template, for all the elements of the collection denoted by the collection displayer (in our cases for all the classes defined in the model). The collection is specified by the following Xion expression:

```
BM::Class.allInstances()
```

`BM` stands for `Business Model`, and is the name of the enclosing package. The `allInstances()` operation applies to the `Class` class, and retrieves all its instances, and returns them as a set.

The following picture shows the text of the `XMI_owned_element` template.

```
<UML:Namespace.ownedElement>
    !-!/objexion/197 AllClasses/
</UML:Namespace.ownedElement>
```

Figure 13 : Template for the generation of the owned element section of the XMI file.

The XMI text for the classes is modelled by the following model.

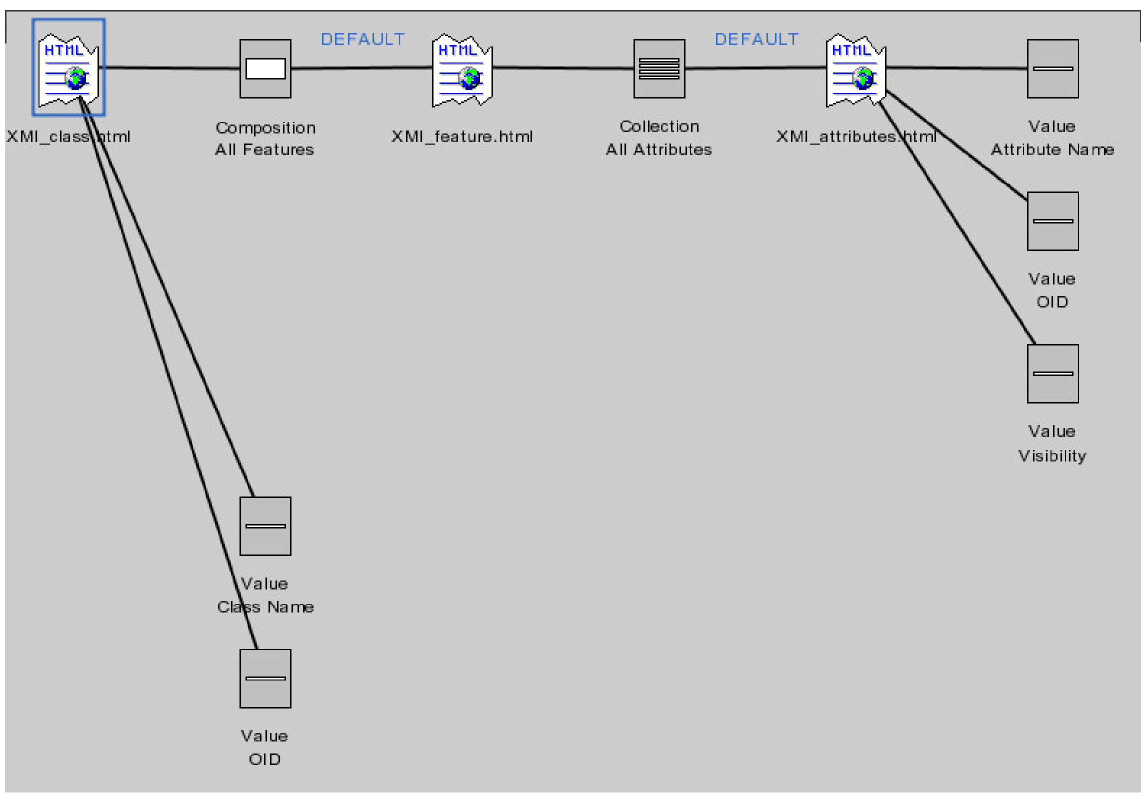

Figure 14 : Model of the class template, with inclusion of the feature template, class name and unique identifier.

The `XMI_class` template contains the text of the class level of the XMI file, inserts the name of the class and its unique identifier, and uses a composition decision center (`All Features`) to embed the feature level of the XMI file.

The following picture shows the template for the class. Notice the three placeholders, to insert the unique identifier, the name of the class, and the various features.

```
<UML:Class xmi.id="!-!/objexion/202 OID/" name="!-!/objexion/201 ClassName/">
    !-!/objexion/198 AllFeatures/
</UML:Class>
```

Figure 15 : Text of the template for the classes.

In our simplified meta-model, attributes are the only features contained by the classes. We generate the feature part of the XMI, by inlining the `XMI_attribute` template for all the attributes retrieved by the collection displayer

The Xion expression resolved by the collection displayer is:

```
c.attribute
```

where `c` is a parameter of type `class`, which was transmitted to the `XMI_template`. As in OCL, `attribute` with a lower case denotes the collection of attributes which are linked to the class specified by `c`.

The XMI text of the feature level template is given in the figure below.

```
<UML:Classifier.feature>
    !-!/objexion/199 AllAttributes/
</UML:Classifier.feature>
```

Figure 16 : Text of the template for the feature level of the XMI file.

The template expansion proceeds further with the expansion of the template for the attributes, for all the attributes contained in the collection. The XMI text of the attribute level template is given in the figure below.

```
<UML:Attribute  xmi.id="!-!/objexion/203 OID/"
                name="!-!/objexion/200 AttributeName/"
                visibility="!-!/objexion/204 Visibility/">
</UML:Attribute>
```

Figure 17 : Text of the template for the attributes.

At this point the generation process of the XMI file is fully specified, and executable code can be generated from the model.

The business and composition models are platform independent models. The Netsilon tool can translate them into several executable platform specific models, including PHP, Java Servlet and JSP application servers.

As an illustration, we give below an excerpt of the PHP code generated for the decision center `AllAttributes` presented earlier which iterates over all the attributes of a class. This code is not intended to be read by humans, and we will only give an overview of it structure.

```
function all_attributes_1(&$ctxt,&$_lc){
      $_t2=new  OclAnySetSql($ctxt,("SELECT  gpattribute.attribute_id  FROM
gpattribute   WHERE   ((gpattribute.attribute_id)   IS   NOT   NULL)   AND
((gpattribute.class) = (".codeOclObject($_lc)."))  ",0,103,FALSE);
      checkNull($ctxt,$_t2,"expression  in  decision  center:  All  Attributes
(199)");
      $_t1=$_t2->elements();
      $_lindex=0;
      while ( $_t1->hasMoreElements() )
            {
                  $_lelement=$_t1->nextElement();
```

```
                        {
                              xmi_attributes_($ctxt,$_lelement);
                        }
                        $_lindex=$_lindex+1;
                  }
}
```

Figure 18 : PHP function generated for the collection displayer AllAttributes.

In this function we can see how the attributes are fetched from the relational database (with the SELECT instruction) and then we see the iteration over all these attributes and the call to the function which handles the attributes: `xmi_attributes_($ctxt,$_lelement);`

Below, we give the generated PHP code for this inline expansion of the attribute template.

```
function xmi_attributes_(&$ctxt,&$_la){
$ctxt->writeToOutputStream("<UML:Attribute xmi.id=\"");
      $_t1=$_la;
      oid_1($ctxt,$_t1);
$ctxt->writeToOutputStream("\" name=\"");
      $_t2=$_la;
      attribute_name_3($ctxt,$_t2);
$ctxt->writeToOutputStream("\" visibility=\"");
      $_t3=$_la;
      visibility_($ctxt,$_t3);
$ctxt->writeToOutputStream("\">\n</UML:Attribute>\n");
```

Figure 19 : PHP function which performs the inline-expansion of the attribute template.

In the next section we briefly describe an example of Java code generation for the same model.

Java code generation

We give below the satellite view of the composition model for Java code generation.

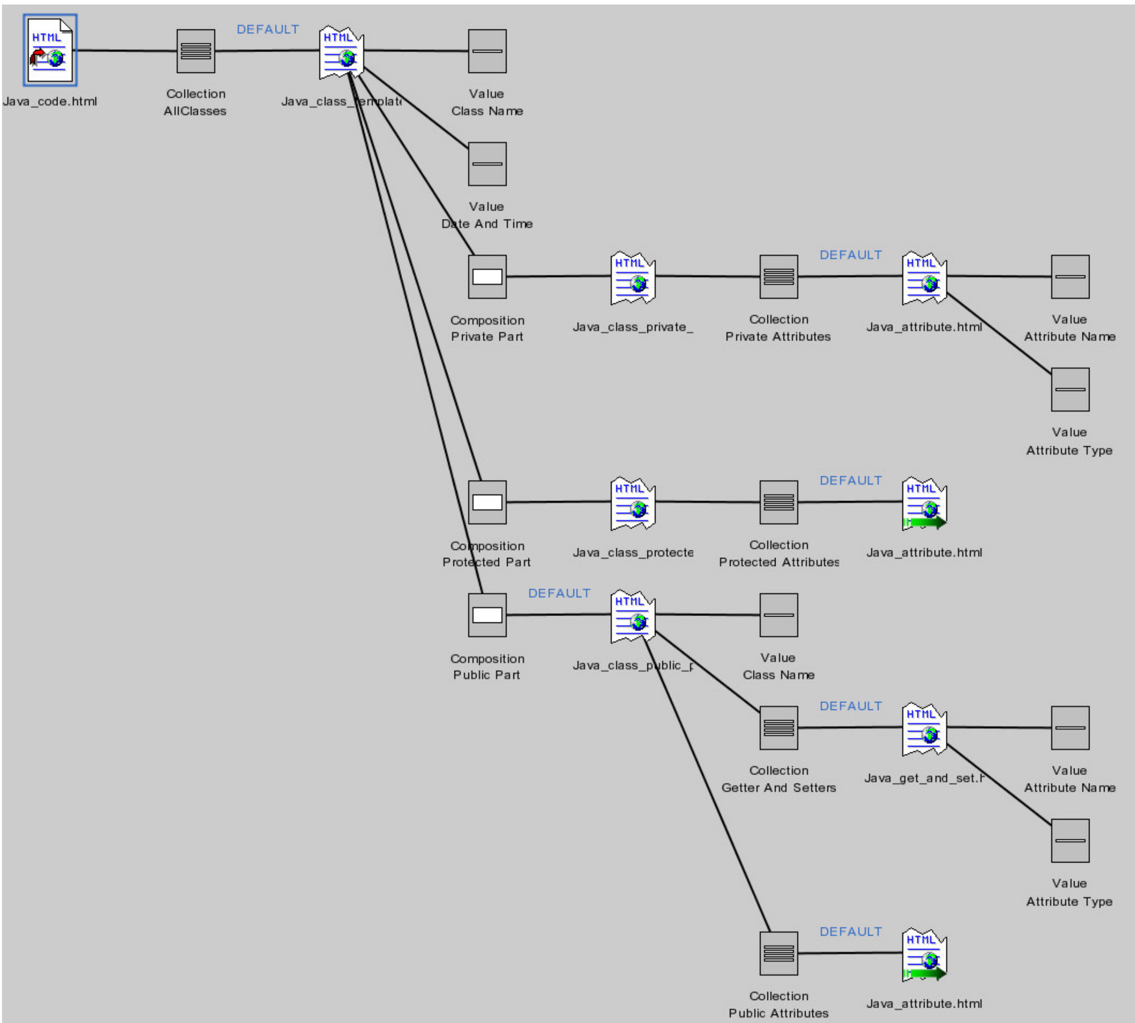

Figure 20 : Satellite view of the composition model for Java code generation

This time we will focus on the generation of alternate constructs, based on the visibility of an attribute. In the business model presented earlier, the `Person` class has got three attributes (`Name`, `Firstname` and `Age`).

The following picture shows the Java code generated when all attributes are public.

```
//
//        generated        the        2004-10-05        at        09:53:27
//
Class                                                              Person
{

      public                                                     Person(){}

      public                          String                          name;
      public                         String                      firstName;
      public int age;

} // end Person
```

Figure 21 : Generated Java code, with public attribute members.

Now we switch the visibility of the `Age` attribute to `private`, and re-execute the generation program. We obtain the following code, with getters and setters operations.

```
//
//        generated        the        2004-10-05        at        09:45:49
//
Class                                                              Person
{

      public                                                     Person(){}

      public                          String                          name;
      public                         String                      firstName;

      public void setAge (int theAge) {
            age = theAge;
      }

      public int getAge() {
            return age;
      }

      private int age;
} // end Person
```

Figure 22 : Generated Java code, with one private attribute member.

In the next paragraphs we will present the composition model which takes care of this generation variation based on the visibility of the attributes.

The following picture presents the composition model of the public part of a Java class. The public part embeds a collection of Getters and Setters (when the class defined non-public attributes) and a collection of public attributes.

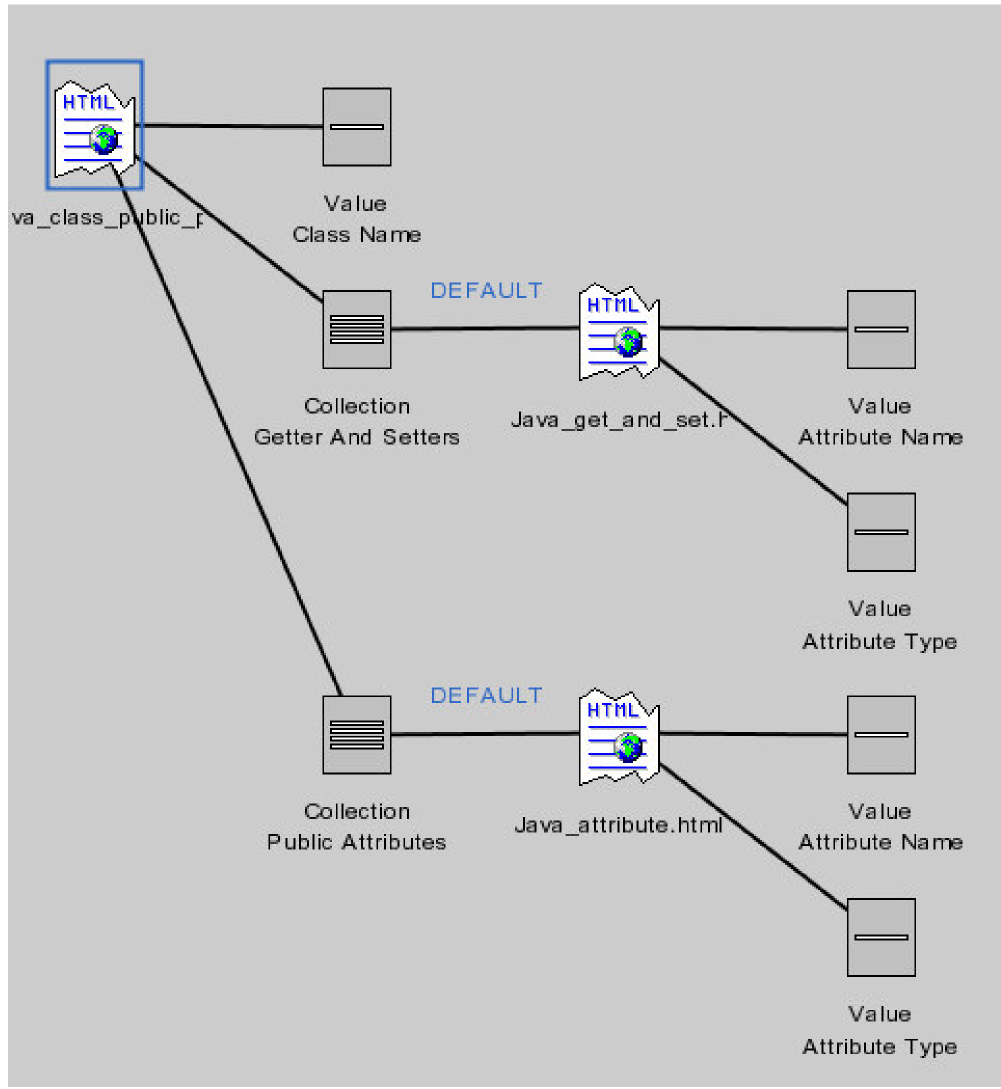

Figure 23 : Composition model for the public part of a Java Class.

The Xion expression contained by the two collection displayers is:

```
c.attribute->select(Visibility != #Public)
```

for the `Getters and Setters` Decision Center, and conversely

```
c.attribute->select(Visibility == #Public)
```

for the `Public Attributes` Decision Center.

The following picture shows the text of the template for the public part of a Java class.

```
<br>
<blockquote>
public !-!/objexion/193 ClassName/(){}<br>
<br>
!-!/objexion/130 PublicAttributes/<br>
!-!/objexion/190 GetterAndSetters/
</blockquote>
```

Figure 24 : Text of the template for the public part of a Java class.

Notice the HTML tags used to specify the final layout of the text. The `<br>` tags force a line break while the `<blockquote>` tags indents the text. The `Value Displayer` is used twice, to generate the names of the default constructors and destructors. Then, the two collection displayers handle the templates for public attributes and getters and setters. Notice that if a collection is empty, as it is the case when there are only public attributes, then the collection displayer inlines nothing.

The next picture shows the text of the template used for the generation of the getters and setters operations. This template will be inline-expanded in the public part of the Java class for all the attributes which have not public visibility.

```
public void set!-!/objexion/191 CapAttributeName/
        (!-!/objexion/192 AttributeType/ the!-!/objexion/191 CapAttributeName/)
{<br>
<blockquote>
!-!/objexion/14 AttributeName/ = the!-!/objexion/191 CapAttributeName/;<br>
</blockquote>
}<br>
<br>
public !-!/objexion/192 AttributeType/ get!-!/objexion/191 CapAttributeName/()
{<br>
<blockquote>
return !-!/objexion/14 AttributeName/;<br>
</blockquote>
}<br>
<br>
```

Figure 25 : Text of the template for getters and setters.

Again, HTML tags format the template text, and value displayers insert the details (`Name` and `Type`) of the current attribute (which is accessible as an input parameter of the template).

References

[1] P.-A. Muller, Ph. Studer, J. Bezivin, *Platform Independent Web Applicaation Modeling*, UML'03, October 2003.

[2] G. Sunyé, A. LeGuennec, and J.-M. Jézéquel, *Using UML action semantics for model execution and transformation*, Information Systems, Elsevier, 27(6):445--457, July 2002.

[3] T. Elrad (moderator), Mehmet Aksit, Gregor Kiczales, Karl Lieberherr and Harold Ossher (panelists), *A Discussion on Aspect-Oriented Programming: Frequently-Asked Questions*, Communications of the ACM, Vol. 44, No. 10, pp. 33-38, October 2001.

[4] C. Atkinson and Al. , *Component-Based Product Line Engineering with UML,* Addison-Wesley Professional; 1st edition (November 15, 2001)