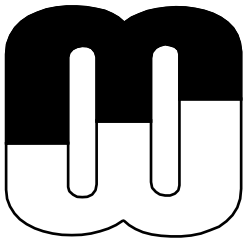

WHITE PAPER

IMPORTING MODELS INTO METAEDIT+ USING XSLT

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä, Finland

Phone +358 400 648 606
Fax +358 420 648 606

E-mail: info@metacase.com
WWW: http://www.metacase.com

IMPORTING MODELS INTO METAEDIT+ USING XSLT

Abstract

MetaEdit+ supports the importation of existing models, thereby enabling a quick and efficient transition from other modeling tools to MetaEdit+. A variety of importation methods are supported which utilize standard formats such as XML and SOAP. This paper describes the use of XSLT for importing models into MetaEdit+.

1 INTRODUCTION TO IMPORTING

There are several ways to import data into MetaEdit+. Figure 1 depicts the main scenarios. All scenarios are numbered and they start from top and move towards the bottom as is indicated by the arrows. Grayed-wheel symbols describe phases where operations (automated or manual) are done to the input data. Blue pentagon symbols illustrate input or output of these operations. Arrow relationships connect these together and represent the execution order.

Starting from the left, scenario 1 presents the reverse engineering option where the transformation is done with MetaEdit+ Reporting Language (MERL). Next three scenarios (scenarios: 2-4) utilize ASCII based text transformation, formatting and importation is accomplished via MetaEdit+'s XML import function, called MXM, just like in scenario 1. This MXM format enables the exchange of XML data between MetaEdit+ environment and other programs. Scenario 5 uses the same ASCII format, but importation is accomplished via MetaEdit+ SOAP API. This API can be also used directly to retrieve data from another tool's repository; Scenario 6 presents this option.

1. Reverse-engineer the code with MetaEdit+ Reporting Language
    a. Tool generate the models to code in ASCII format, then MetaEdit+ Reporting Language is used to transform the models to MXM format
2. Other tool exports directly to MetaEdit+ XML format (MXM)
    a. This is preferable approach if the other tool can export models in user-definable formats.
3. Export to other tool that can then export directly to MXM
    a. If the “source tool” is not powerful enough for customizing the export format, you may consider transforming models in the same format (e.g. XMI) with a more powerful 3rd party tools and then use option 2.
4. Use XSLT to transform the models to MetaEdit+ XML
    a. Tool exports models in XML/XMI format, then XSLT transformation language is used to transform the models to MXM format

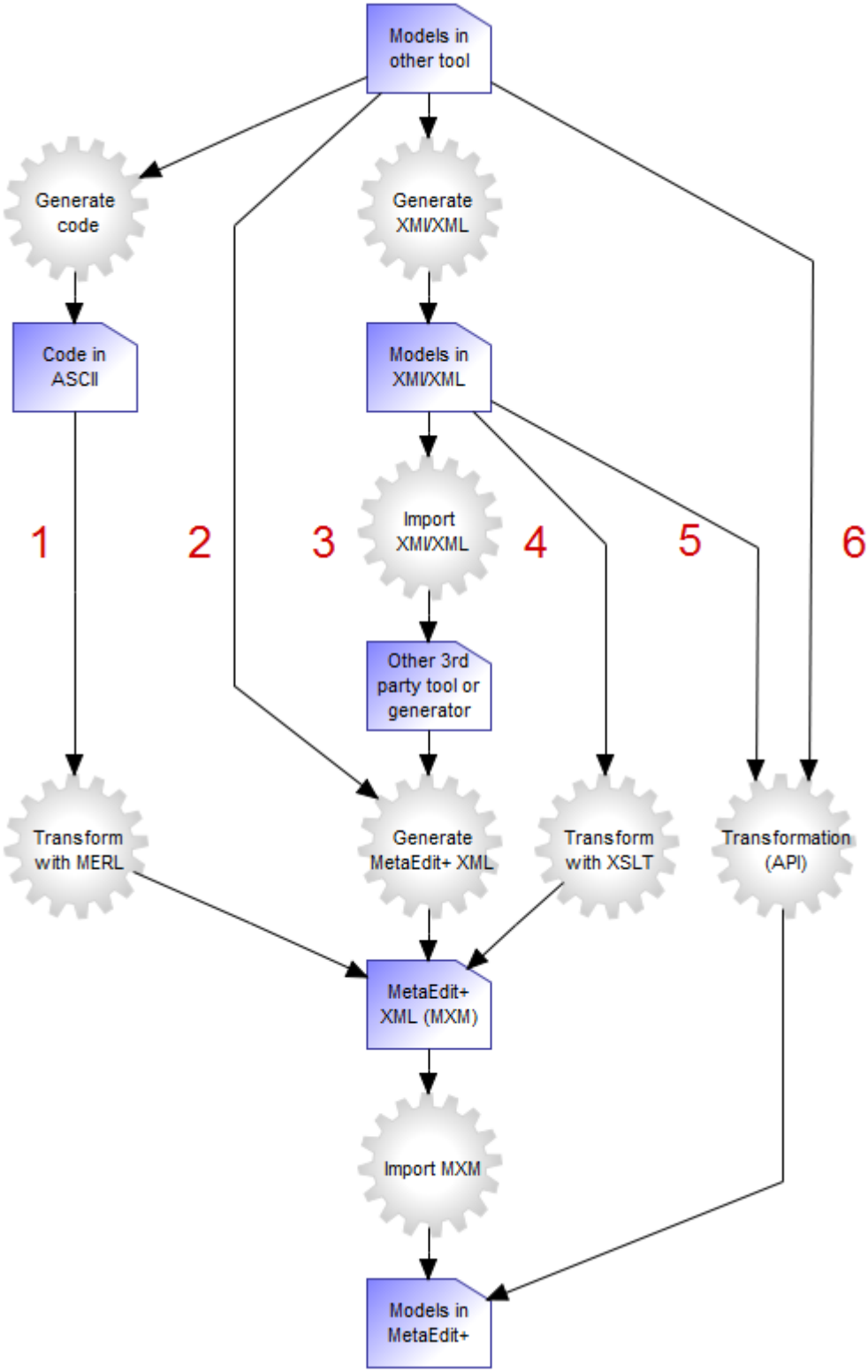

Figure 1 Importing scenarios

5. A program transforms the exported models (e.g. in XML files) and accesses MetaEdit+ programmatically
    a. The transformation can be made in your preferable language since MetaEdit+ SOAP API supports almost any programming language
6. Other tool accesses MetaEdit+ programmatically
    a. E.g. an Office tool creates a link directly to MetaEdit+

In this project we utilized Scenario 4 and used a generic XML tool to transform the data with the XSLT script. For other options please contact info@metacase.com.

2 XSLT TRANSFORMATION STEPS

The main phases in the XSLT transformation are as follows:

- Export models as XML file; a sample of the exported data is shown in Figure 2
- Implement the XSLT transformation script which takes the generated data as an input and produces the required format as an output (MetaEdit+ MXM file)
- Import the generated data consisting of transformed models
- View models in MetaEdit+; fine tuning models by hand or with auto-layout functionality

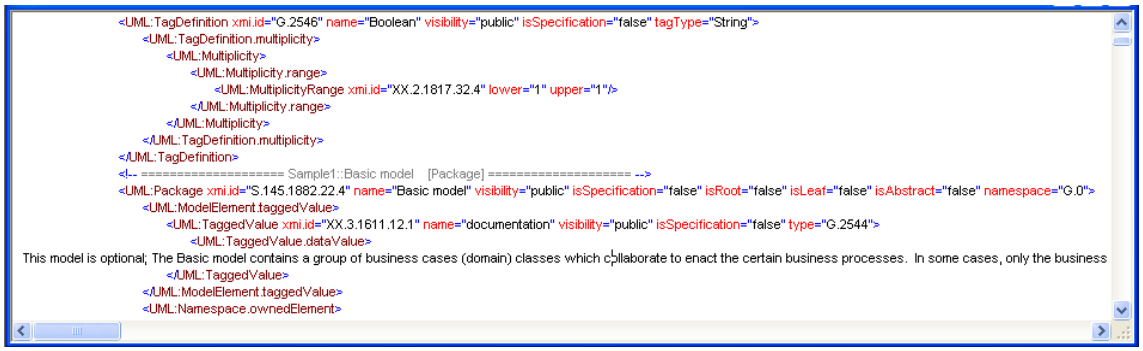

Figure 2 Exported model in XML format (UML model)

To be able to have the same models in MetaEdit+, all models need to follow MetaEdit+'s metamodel definitions. In Figure 3 below, a portion of MetaEdit+'s UML metamodel is represented in XML format:

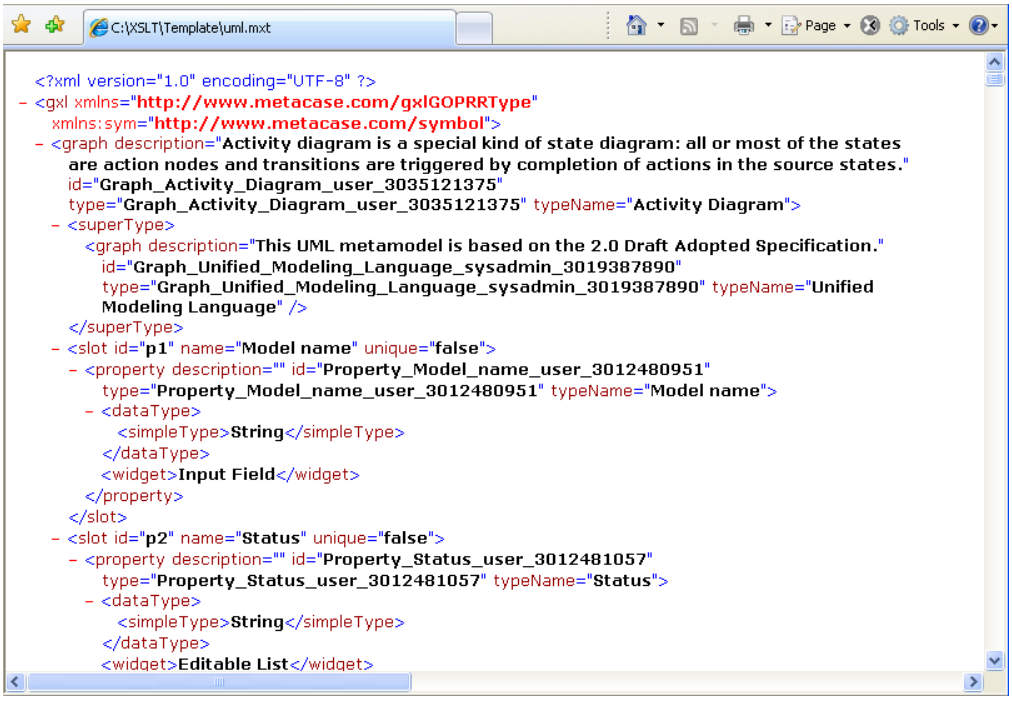

Figure 3 UML metamodel in XML format

In this case, the metamodel is the same for both tools: we export UML models based on XMI and then transform them into UML models MXM. For the transformation we wrote XSLT script and associate it to the XMI file presented in Figure 2. In the next section we inspect the XSLT script in more detail.

The transformation is completed by associating and running the XSLT script with the XMI file in your chosen XML editor. In Figure 4, the generated MXM file is depicted. It includes data for UML models and also possibly representations such as size, position, and scaling. The file extension used here is .mxm and it stands for MetaEdit+ XML Models. You can also transform metamodels in XML format using .mxt (MetaEdit+ XML Types).

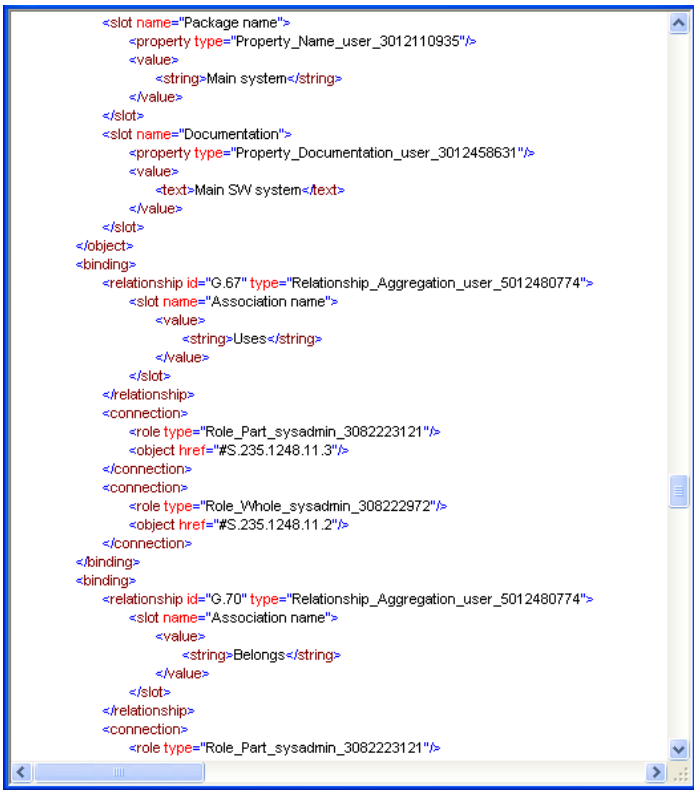

Figure 4 Resulting MXM file ready for importation

3 XSLT SCRIPT IN MORE DETAIL

The XSLT script can be divided into the sections listed below:

- Looping through all "UML:Model" elements in the source XML file
- Creating the model's conceptual information into MXM file
- For each object type in the XMI file (Package, Class, Component, Interface, etc.)
    - Create object's information—each object type has its own template in the XSLT script since for example classes are transformed differently than packages. Since the object type have some commonalities you may apply reusable templates—e.g. documentation property exists in several places or finding the right stereotype value, were having own reusable templates in XSLT script.

    - Check for possibly owned elements that link the objects to other elements. For example, if owned elements are found for a package there needs to be a sub graph hierarchy in MXM,
- Add packages and models which have a diagram representation (can be found from own element tag in the source XML).
- Create the objects and their representational data
    - Calculate the positions, scaling, etc.
    - Create relationships (aggregations, associations, generalizations, dependencies)

In Listing 1, a short sample of the UML Class template XSLT script is presented:

Lines 1-3; comment for the Class template

Line 4; begins the template and matches it to the structure 'UML:Classes' found under the 'UML:Namespace.ownedElement' tag

Line 5; produces the object's opening tag

Line 6-7; adds the 'id' and 'type' attributes with the correct value to the object tag

Lines 8-15; produces the 'slot' element's contents; first comes the name, then comes the property tag with the type attribute value, and last comes the value tag with the data type (string tags) and the class name which is fetched from the source document's 'name' attribute (line 12). After that, all the elements are closed.

Lines 16-20; start to produce the Attribute's contents within the similar manner as above

Lines 21-23; goes through all 'UML:Classifier.features' in the loop and calls an own template for each 'UML:Attribute' found in the source document

Lines 24-26; closes the open tags in the target document

```
1  <!-- ############### -->
2  <!-- Class template -->
3  <!-- ############### -->
4  <xsl:template match="UML:Namespace.ownedElement/UML:Class">
5    <object>
6      <xsl:attribute name="id"><xsl:value-of select="@xmi.id"/></xsl:attribute>
7      <xsl:attribute name="type">Object_Class__UML__sysadmin_3019392012</xsl:attribute>
8      <slot name="Class name">
9        <property type="Property_Class_name_user_3012480910"/>
10       <value>
11         <string>
12           <xsl:value-of select="@name"/>
13         </string>
14       </value>
15     </slot>
16     <!-- Attributes -->
17     <slot name="Attributes">
18       <property type="Property_Attributes__UML__sysadmin_3019391069"/>
19       <value>
20         <seq>
21           <xsl:for-each select="UML:Classifier.feature">
22             <xsl:apply-templates select="UML:Attribute"/>
23           </xsl:for-each>
24         </seq>
25       </value>
26     </slot>
```

Listing 1 Class template's XSLT script

4 CONCLUDING REMARKS

Models (and metamodels) can be exported into MetaEdit+ from different sources. We described here the principle of using XSLT to import models from another tool. If you plan to implement XSLT transformation you need to recognize that:

- Metamodels may differ between the tools (even if they are UML), and export functionality may have several exporting options; concept(s), structures (e.g. relationship types, multiplicity & navigable property fields), and object properties.
    - This had to be first checked and then determined how to handle the differences and inconsistencies.
- In the case of XMI, there exist different versions and generally it does not work between different tools for other than simple cases.
- Representational data differs between the tools.
    - Having a reasonable scaling and resizing factor in use during the conversion.
- To have instances appear only once in the target repository, i.e. have the correct reuse for the design elements, the concept needed to be introduced once and later only referenced.