ITEA Office
High Tech Campus 69 - 3
5656 AG Eindhoven
The Netherlands

T + 31 88 003 6136
E info@itea3.org
W www.itea3.org

ITEA 3 is a EUREKA strategic ICT cluster programme

BUMBLE Deliverable D3.5 (Version 1)

MPS-based Blended Modeling Generation Environment

Edited by: BUMBLE Team
Date: November 2021

Project: BUMBLE - Blended Modeling for Enhanced Software and Systems Engineering

Contents

Acronyms

AST	Abstract Syntax Tree
DSML	Domain Specific Modelling Language
EMF	Eclipse Modelling Framework
MPS	Meta Programming System
UC	Use Case
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1. Introduction

This document describes the initial results for model generation environments that exploit Jetbrains MPS as core DSML technology. Apart from the out-of-the-box facility in MPS to import EMF models as described in Deliverable D3.2, BUMBLE also exploits some activities in the context of UC1 and UC5 where MPS models are generated. Only these use cases involve generation of model environments beyond the normal use of Jetbrains MPS. The work on such generation in the context of UC5 has however not yet started at the time of writing this version of this deliverable. Hence, we only report on the work performed in the context of UC1.

The remainder of this deliverable is structured as follows. Chapter 2 describes an approach to generate models in the context of mapping meta models (DSML definitions) between EMF and Jetbrains MPS. Table 1 provides links to download relevant sources. The links for UC5 refer to the source code for DClare and DClareForMPS, which is the context for the involved work to start in UC5.

Table 1. Links for downloading open-source solutions described in this deliverable.

Use Case	Chapter	Links
UC1	2	https://github.com/hilalosoft/exchanging_ecore_model_MPS (code) https://play.mdh.se/media/t/0_4qpus1y0 (video)
UC5	-	https://github.com/ModelingValueGroup/dclare (code) https://github.com/ModelingValueGroup/dclareForMPS (code)

2. EMF/MPS Interoperability (UC1)

This chapter discusses the BUMBLE efforts in UC1 on bridging two language workbenches, JetBrains MPS (later also referred to as simply MPS) and the Eclipse Modeling Framework (EMF). More specifically, the work is devoted to the mapping of metamodels defined in MPS towards metamodels conforming to the EMF specification language, namely Ecore. Technically, we firstly contribute with an Ecore language specification for MPS. Based on this, users can create metamodels by using the MPS language workbench features and possibly create models conforming to such metamodels. Alternatively, we contribute with a transformation for mapping the metamodels defined in MPS as Ecore metamodels usable in EMF. In this latter scenario, users can leverage EMF plug-ins e.g., to generate default tree editors, create a custom concrete syntax, use the metamodel as part of a model transformation chain, and so forth.

2.1. Bridging MPS and EMF

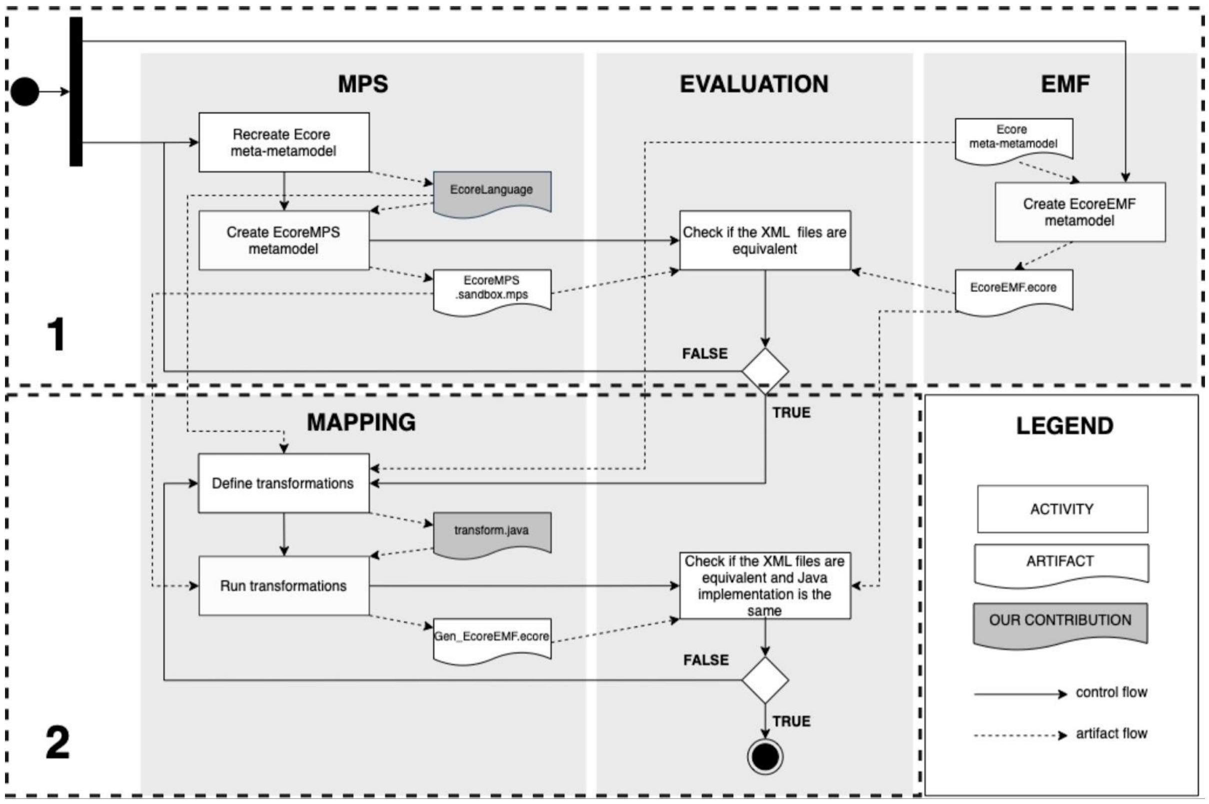

Figure 1. Process of bridging MPS and EMF, and evaluation of the solution.

Throughout the rest of the text, we refer to metamodels created in EMF as EcoreEMF metamodels (*.ecore* extension), and the ones created in MPS as EcoreMPS metamodels (*.sandbox.mps* extension). Implementation and evaluation of this solution are carried out by following the process illustrated in Figure 1. The process consists of two main steps.

STEP 1: Recreate the Ecore meta-metamodel, as defined in EMF, as a language (EcoreLanguage) in MPS and evaluate whether the EcoreMPS metamodel conforming to EcoreLanguage is equivalent to the EcoreEMF metamodel conforming to the Ecore meta-metamodel in EMF. By equivalent, we mean that the metamodels contain the same concepts and hierarchical structure, while the order by which metaconcepts persist in the metamodel file might differ.

STEP 2: Define automated mechanisms (i.e., model transformations) to transform EcoreMPS metamodels to Gen_EcoreEMF metamodels and evaluate whether the latter is equivalent to the corresponding EcoreEMF metamodel and whether it can be correctly loaded and used in EMF. In addition, the generated Java classes from both metamodels (i.e., Gen_EcoreEMF metamodel and EcoreEMF metamodel), should be the same.

EcoreLanguage: Implementing Ecore in MPS
The first step towards providing a bridge between EMF and MPS is recreating the Ecore meta-metamodel as an EcoreLanguage in MPS. The structure of EcoreLanguage consists of concepts, concept interfaces, and their corresponding children, properties, and references, as found in the Ecore meta-metamodel. For each concept of the language, there is an editor that facilitates the manipulation of the Abstract Syntax Tree (AST) and provides intuitive interaction. The constraints aspect is used to express advanced constraints that cannot be covered by the language structure. Moreover, to allow the initialization of some properties/references/children to default values when a concept instance is created, we use concept constructors and rely on the behavior language aspect in MPS. Upon complete implementation, the language is packaged as a plugin that can be distributed to users. By importing this language, we can start defining EcoreMPS metamodels.

Automatic Export from MPS to EMF
To build a bridge and enable the exchange of metamodels between MPS and EMF, EcoreMPS metamodels are transformed to Gen_EcoreEMF metamodels that can be correctly loaded and used in EMF. The transformations are defined in Java and driven by an implicit mapping that is used to define correspondences between elements of the source (i.e., EcoreLanguage) and target (i.e., Ecore meta-metamodel) languages. While defining these correspondences, it is important to fully understand the structure of both languages, thus, in the following we provide code excerpts from the definition of a metamodel, both in EMF and MPS. To simplify the reading, we describe the procedure by its instantiation on a specific example, the Family metamodel (depicted in Figure 2 in terms of EcoreEMF).

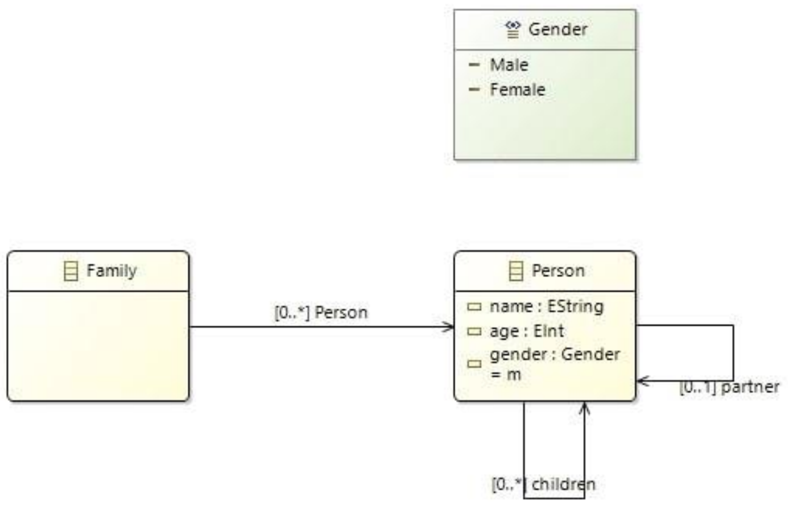

Figure 2. Family EcoreEMF metamodel.

The listing in Table 2 details the eClassifier `Gender` of type EEnum, whose values are restricted to eLiterals `Male` and `Female` from the *family.ecore* file. If we make a reference to Figure 1, *family.ecore* represents the *EcoreEMF.ecore* artifact.

Table 2. XML version of family.ecore.

```
1.  <eClassifiers name="Gender" xsi:type="ecore:EEnum">
2.      <eLiterals literal="m" name="Male" value="0"/>
3.      <eLiterals name="Female" value="1"/>
4.  </eClassifiers>
```

The listing in Table 3 details the `Gender` concept, from the *familymodel.sandbox.mps* file that represents the *EcoreMPS.sandbox.mps* artifact in Figure 1 and it consists of two parts.

1. Language definition: Defines the languages used for the definition of EcoreMPS metamodels. For this specific case, we have used a built-in language from MPS (line 3-5), and EcoreLanguage (line 6-11). EcoreLanguage reflects the Ecore meta-metamodel, where each concept can contain references, properties, and children, and they are all assigned randomly generated index values. Line 7 details the `EEnum` concept, while line 10 the `EEnumLiteral` concept.

Table 3. XML version of family.sandbox.mps.

```
1.  <-- language definition -->
2.  <registry>
3.      <language id="ceab5195-25ea-4f22-9b92-103b95ca8c0c"
                        name="jetbrains.mps.lang.core">
4.          ...
5.      </language>
6.      <language id="45e9c502-be8d-4b95-92c9-8ad2f7c494aa"
                        name="EcoreLanguage">
7.          <concept id="5921274573544802721"
                        name="EcoreLanguage.structure.EEnum" flags="ng"
                        index="1BB5TV">
8.              <child id="5921274573544802722" name="eLiterals"
                        index="1BB5TS"/>
9.          </concept>
10.         <concept id="5921274573544831328"
                        name="EcoreLanguage.structure.EEnumLiteral"
                        flags="ng" index="1BBqUU"/>
11.     </language>
12.</registry>

13.<-- metamodel definition -->
14.<node concept="1BB5TV" id="5qTU7U3AdSP" role="3Lc43O">
15.     <property role="TrG5h" value="Gender"/>
16.     <node concept="1BBqUU" id="5qTU7U3AdT0" role="1BB5TS">
17.         <property role="TrG5h" value="Male"/>
18.         <property role="1BBqUN" value="0"/>
19.         <property role="1BBqUK" value="male"/>
20.     </node>
21.     <node concept="1BBqUU" id="5qTU7U3AdT4" role="1BB5TS">
22.         <property role="TrG5h" value="Female"/>
23.         <property role="1BBqUN" value="1"/>
24.         <property role="1BBqUK" value="female"/>
25.     </node>
26. </node>
```

2. Metamodel description: Defines the instances of concepts (i.e., nodes) that are described in the language. Each node stores a reference to its declaration, its concept. The node in line 14 stores a reference to `EEnum`, while the node in line 16, stores a reference to `EEnumLiteral`.

To connect the language definition and metamodel definition, we use hash-tables as data structures that can map keys to values. Starting from the language definition, we iterate through all the elements of the language, and store the element's `index` as key, and the element's `name` as value. Being that nodes store references in the attribute `concept`, while properties and references store references in the attribute `role`, we define two hash-tables; one for conceptElements and one for propertyElements (used both for properties and references).

The next step consists in defining a recursive function that leverages the tree-like structure of XMI files and traverses the nodes, starting from the root node in the XMI file and branching to the leaves. First, we access the `get(Object key)` method of the conceptElements hash-table, which returns the value to which the key is mapped in this hash-table. Depending on the meta-object that equals the returned value, the implementation outputs an XML file that follows the same template as an EcoreEMF metamodel.

The listing in Table 4 details a code excerpt of the analyzeNode function. In lines 2 and 3, we have conditional statements that perform different computations, depending on the meta object that equals the returned value of the `get()` method. In the listing we only illustrate the “EPackage” and “EEnum” meta objects. Lines 4-13 detail the computations that take place when the conditional statement that checks whether the returned value of the `get()` method equals “EEnum”, evaluates to true. The first step in this computation consists in creating an *eClassifier* and adding the attribute that identifies an EEnum in XMI to it. Next, we iterate through the childNodes of the node we are currently analyzing.

Table 4. analyzeNode function.

```
1. public static Node analyzeNode(Node concept) {
2.     if (conceptElements.get(concept.getAttributes().
                   getNamedItem("concept"). getNodeValue()).
                   equals("EPackage")) {...}
3.     else if (conceptElements.get(concept.getAttributes().
                   getNamedItem("concept"). getNodeValue()).
                   equals("EEnum")) {
4.         element = eclipseEcoreXML.createElement("eClassifiers");
5.         element.setAttribute("xsi:type","ecore:EEnum");
6.         for (int i=1; i <concept.getChildNodes().getLength(); i=i+2) {
7.             if (concept.getChildNodes().item(i).getNodeName().
                   equals("node")) {
8.                 element.appendChild(analyzeNode(concept.
                   getChildNodes().item(i)));
9.             } else if (propertyElements.get(concept.getChildNodes().
                   item(i).getAttributes().getNamedItem("role").
                   getNodeValue()).equals("name")) {
10.                element.setAttribute("name",concept.getChildNodes().
                   item(i).getAttributes().getNamedItem("value").
                   getNodeValue());
11.            }
12.        }
13.        return element;
14.    }
15.    …
16.}
```

If the childNode's name is equal to “node”, the childNode is passed as a parameter to the analyzeNode function, and the `appendChild()` method is used to append this childNode to the list of children of the node under analysis. Else, we access the `get()` method of the propertyElements hashtable, where we pass as a parameter the value of the childNode's `role` attribute. If the returned value equals to “name”, then we set the value of the childNode's `value` attribute on the `name` attribute of the element.

After all the nodes of the XMI file are visited, we run the transformations, which use the family.sandbox.mps file (corresponds to EcoreMPS.sandbox.mps) as input and generate the family.ecore file (corresponds to Gen_EcoreEMF.ecore) described in the listing of Table 5 as output.

Table 5. XML version of the generated family.ecore

```
<eClassifiers xsi:type="ecore:EEnum" name="Gender">
    <eLiterals name="Male"/>
    <eLiterals name="Female" value="1"/>
</eClassifiers>
```

2.2. Evaluation

The evaluation process of the solution conceives of two major steps, as illustrated in Figure 1:

STEP 1: Concerns the correctness of EcoreLanguage, which is validated via conceptual and structural comparison of EcoreMPS and EcoreEMF versions of a same metamodel. In case we identify inconsistencies between the two metamodels, EcoreLanguage is adjusted accordingly. The advantage of this evaluation step is two-fold. First, it assures that the EcoreLanguage is well-defined and the artefacts that are used as input to the transformations are correct. Second, validating the implementation in an iterative manner reduces time and effort in case of errors, as it facilitates the identification of the erroneous artefact. If the evaluation were only performed at the end of STEP 2, it would be extremely challenging to identify the erroneous artefact (i.e., EcoreLanguage or transformations). With regards to our example, we needed to compare the metamodel EMF definition in the listing of Table 2 to the metamodel MPS definition in the listing of Table 3. Both metamodels include the EEnum Gender that contains two ELiterals (Male and Female) as children, thus we consider them equivalent, as they contain the same concepts and hierarchical structure.

STEP 2: Focuses on the correctness of the transformation implementation. For the transformations to be considered correct, the following conditions need to be fulfilled: i) the Gen_EcoreEMF metamodel should be correctly loaded in EMF, ii) the XMI of Gen_EcoreEMF metamodel and the XMI of EcoreEMF metamodel need to be equivalent, and iii) the generated Java classes from the genmodel of each metamodel need to be equivalent. If any of these conditions is not fulfilled, the transformations need refinement. Considering the Family metamodel, we needed to compare the listing in Table 2 of that in Table 5. As it can be seen, both listings contain the same eClassifiers and eLiteral, as well as the same structural hierarchy. The order of elements and attributes might differ, but that does not affect the output, since the generated Java classes (implementing the metamodel in the modelling ecosystem as editors and resources) are the same for both metamodels.

It is important to emphasize that, although in the described example model, we leverage the sole Family metamodel for exemplification purposes, in the actual evaluation process it was only the simplest metamodel that we accounted for several metamodels (e.g., SmartHome and Airport), with varying complexity in terms of number of meta elements, were used for evaluation purposes too.

The interested reader can download the open-source implementation at https://github.com/hilalosoft/exchanging_ecore_model_MPS as well as watch a demo of the solution at work at https://play.mdh.se/media/t/0_4qpus1y0.