Modeling the "Ecore to GenModel" Transformation with EMF Henshin

Enrico Biermann[1], Claudia Ermel[1], and Stefan Jurack[2]

[1] Technische Universität Berlin, Germany
{enrico,lieske}@cs.tu-berlin.de

[2] Universität Marburg, Germany
sjurack@informatik.uni-marburg.de

Abstract. Our recently developed tool Henshin is an Eclipse plug-in supporting visual modeling and execution of rule-based EMF model transformations. In this paper we describe how we use Henshin to define visual EMF model transformation rules and control structures transforming an Ecore meta-model to a GenModel (case study 3 of TTC 2010). For validation, the model transformation is applied to the Ecore model of a flowchart language.

1 Introduction: Transforming Ecore to GenModel

The most important benefit of the Eclipse Modeling Framework EMF is its ability to generate code automatically. Most of the data needed by the EMF generator for generating code is stored in the Ecore model, e.g. the classes to be generated and their names, attributes, and references. There is, however, more information that needs to be provided to the generator, such as where to put the generated code and what prefix to use for the generated factory and package class names, that is not stored in the core model. The EMF code generator uses a particular EMF model, the *generator model* to get this information. The generator model provides access to all data needed for generation, including the Ecore part, by wrapping the corresponding Ecore model. For example, class `GenClass` wraps (or decorates) `EClass`, class `GenFeature` decorates `EAttribute` and `EReference`, and so on. The EMF generator runs off of a generator model instead of a core model; thus, when using the generator, there are two model resources (files) in the project: a `.ecore` file and a `.genmodel` file. The `.ecore` file is an XMI serialization of the Ecore model and the `.genmodel` file is a serialized generator model with cross-document references to the .ecore file.

Separating the generator model from the Ecore model like this has the advantage that the actual Ecore meta-model can remain pure and independent of any information that is only relevant for code generation. The disadvantage of not storing all the information right in the core model is that a generator model may get out of sync if the referenced core model changes. To handle this, the generator model plug-in offers a facility to reconcile a generator model according to changes made in its corresponding core model without loosing generator-related information.

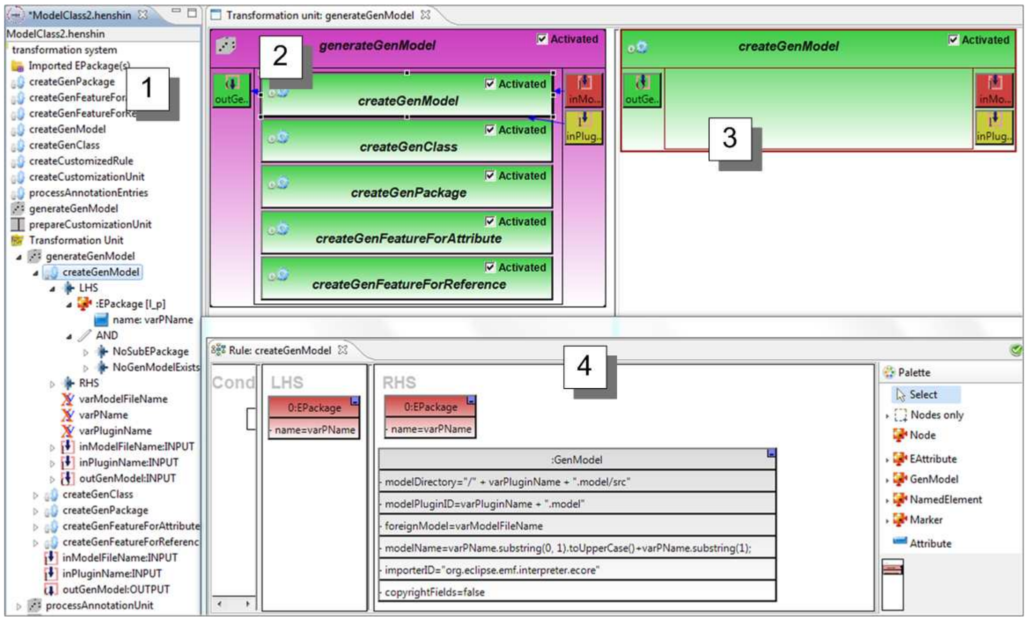
2 Transformation Concepts of Henshin

The transformation approach we use in this paper is based on graph transformation concepts which are lifted to EMF model transformation by also taking containment relations in meta-models into account. Our recently developed tool Henshin[3] is an Eclipse plug-in supporting visual

[3] http://www.eclipse.org/modeling/emft/henshin/, originating from EMF Tiger [1,2,3]

modeling and execution of EMF model transformations based on structured data models and graph transformation concepts.

In our approach, we use the original EMF meta-models Ecore and GenModel as source and target language. In order to support our transformation rules, relations between source and target EMF models are given in a self-provided EMF model *Ecore2Gen*, the so-called mapping model. Apart from defining rules, we made use of the control structures offered by Henshin (called *transformation units*), e.g. constructs for non-deterministic rule choices, rule sequences or rule priority. Those constructs may be nested arbitrarily to define more complex control structures. Passing of model elements and parameters from one rule to another is also possible by using input and output ports. EMF transformation rule applications in Henshin change an EMF instance model in-place, i.e. an EMF instance model is modified directly. Moreover, the pre-definition of (parts of) the match is also supported by Henshin. Henshin currently consists of a *graphical editor* for visually defining EMF model transformation rules and units, and a transformation engine for executing rules and units on EMF models. The transformation engine provides classes which can freely be integrated into existing Java projects which rely on EMF models. Currently there exist two implementations of the transformation engine. One is written in Java while the other translates the transformation rules to Agg [4]. This is useful for validation of consistent EMF model transformations which behave like algebraic graph transformations, e.g. to show functional behavior and correctness [5].



```
modelDirectory="/" + varPluginName + ".model/src"
modelPluginID=varPluginName + ".model"
foreignModel=varModelFileName
modelName=varPName.substring(0, 1).toUpperCase()+varPName.substring(1);
importerID="org.eclipse.emf.interpreter.ecore"
copyrightFields=false
```

Fig. 1. Henshin GUI with tree view (1), transformation unit editor (2) and (3), and rule editor (4).

Fig. 1 shows the preliminary GUI of our Henshin tool. The tree view $\boxed{1}$ allows the modeler to define the needed EPackages for source, target and mapping models of the transformation and the Henshin model itself. Moreover, new rules and transformation units can be created here.

Transformation units can be defined in a visual editor $\boxed{2}$ and may be of type *IndependentUnit* (all contained units are applied in arbitrary order), *SequentialUnit* (all its units are applied sequentially), *CountedUnit* (its units are applied sequentially, each a given number of times), *PriorityUnit* (a child unit of highest priority is applied next) and *AmalgamatedUnit* (for transforming multi-object structures in one step where the number of actually occurring object structures in the instance model is variable). The transformation unit shown in Fig. 1 $\boxed{2}$ is an IndependentUnit (symbolized by a die as icon in the upper left corner) which contains rules as child units. The unit has two input ports and one output port. When the uppermost child unit (rule *createGenModel*) is double-clicked, a view for this unit opens $\boxed{3}$ showing its own child units and its ports. Since rule *createGenModel* has no further child units, this compartment in $\boxed{3}$ is empty. However, colors of the ports of rule createGenModel indicate a connection to ports of its parent unit. The rule view $\boxed{4}$ shows the visual rule editor which comprises three parts for the left-hand side `LHS`, the right-hand side `RHS` and optional conditions `Cond` restricting matches into instance models.

HENSHIN rules and transformation units can be used in other Java projects by instantiating the class `RuleApplication` or `UnitApplication`, respectively. The class `RuleApplication` requires a `Rule` instance from the HENSHIN meta-model. Once instantiated, the rule can be applied by calling the `execute()`-method of `RuleApplication`. Transformation units can be executed in a similar way by using the class `UnitApplication`.

3 The Ecore2GenModel Transformation

Our mapping model combining the source EMF model *Ecore* and the target EMF model *GenModel* is illustrated (without attributes) in Fig. 2. The left-hand side of Fig. 2 shows the *Ecore* model, the right-hand side shows the *GenModel* model, and classes of type `Rel` in between map the corresponding structures of both models.

An EMF model conforming to the Ecore meta-model is now translated by applying the rules in the independent unit `generateGenModel` (see Fig. 1, $\boxed{2}$). In the very beginning, only rule `createGenModel` is applicable (see Fig. 1, $\boxed{4}$). The rule has a nested application condition. The structure of this condition can be seen in the tree view in Fig. 1, $\boxed{1}$, where below the LHS part of rule `createGenModel`, there is an AND node connecting two application conditions (graph constraints on the rule's LHS) which require that there are no super-packages of the EPackage in the LHS and that there is no GenModel existing already. The rule creates a new GenModel node with default values for various attributes. Similarly, GenModel structures are created for EClasses, EPackages, EAttributes and EReferences by applying rules `createGenClass, createGenPackage, createGenFeatureForAttributes` and `createGenFeatureForReference`. Screenshots of these rules contained in unit `generateGenModel` can be found in Appendix B.

Our model transformation transforming an Ecore model to a GenModel (without annotations yet) is applied exemplarily to an Ecore model of a flowchart language[4] from within a Java application by a call to the main transformation unit *generateGenModel's* `execute` method with the source model's file and its URI as input parameters (see lines 89–91 in the complete listing of the Java class file in Appendix A).

[4] `http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/cases/ttc2010_attachment_5_v2010-04-15.zip`

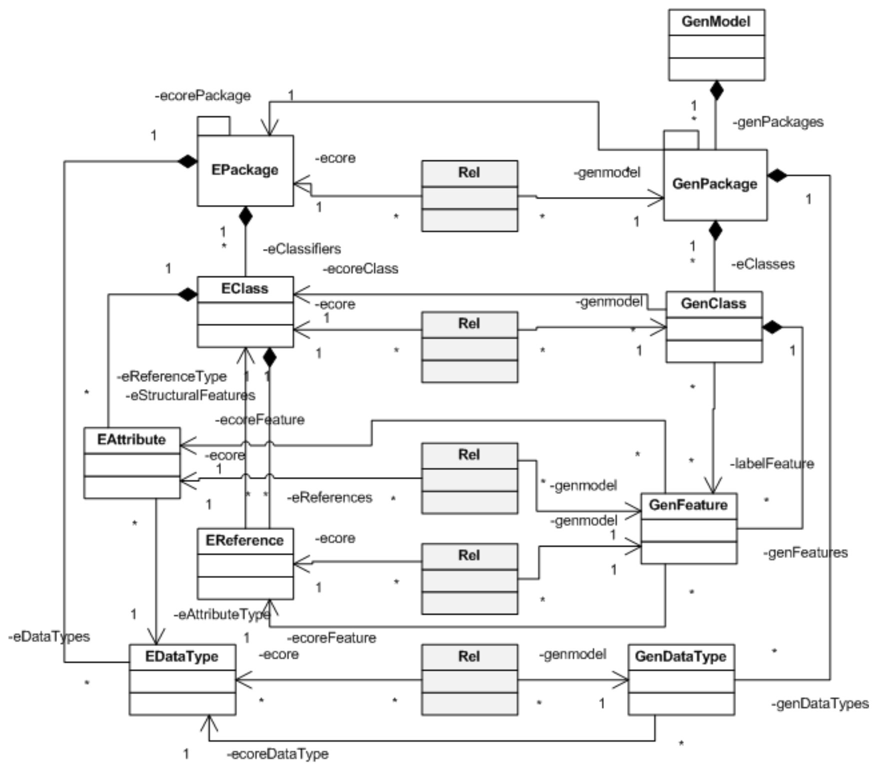


Fig. 2. A part of the mapping meta model for the *Ecore2Genmodel* transformation

4 Extension 2: Transforming GenModel annotations in the source Ecore model by using reflection

We deal with this task by making use of HENSHIN's ability to create a transformation rule by applying another transformation rule. This is a sort of reflection mechanism in HENSHIN which is possible because the HENSHIN transformation system, i.e. rules, transformation units and so one, are defined by an Ecore model. Hence, transformation rules can be applied also to HENSHIN instance models, i.e. to transformation systems and structures within transformation systems such as rules. Depending on the annotations in the source Ecore model, in a first step we generate a customized transformation rule which is tailored to the type of attributes used in the annotation to be processed. In the second step, we apply this customized rule and change the GenModel accordingly by setting the value of the particular attribute in the corresponding GenModel class.

Fig. 3 shows the main unit `prepareCustomizationUnit` to be executed for realizing the extended transformation. Rule `createCustomizationUnit` is called once and creates a container (a *SequentialUnit*) for the customized rule (see Fig. 3). Unit `singleProcessUnit` is applied as long as possible (collecting all `EAnnotations`) and contains two rules to be applied sequentially: rule `processAnnotationEntries` looking for an `EAnnotation` (connected to a class `EStringToString-MapEntry` which contains a `(key, value)` pair of an attribute type and its value) in the Ecore model. The `(key, value)` data together with two more parameters `genType` and `UId` become input parameters to rule `createCustomizedRule`. The input parameter `UId` is an attribute of the `Rel` node connecting the `EModelElement` to the `GenModel` element. The parameter `genType` denotes the type name of the `GenModel` class (e.g. `"GenClass"`, `"GenPackage"` or `"GenFeature"`) the created customized rule is supposed to match. With the help of the input parameters `key` and `value`, the generated rule is able to select the attribute with name `key` and to set its value to `value`.

All rules are shown in detail in Appendix C. In our Java application we first execute the main transformation unit `prepareCustomizationUnit` (see lines 97–101 in the listing in Appendix A), and afterwards apply the generated rules (see lines 103-108 in Appendix A).

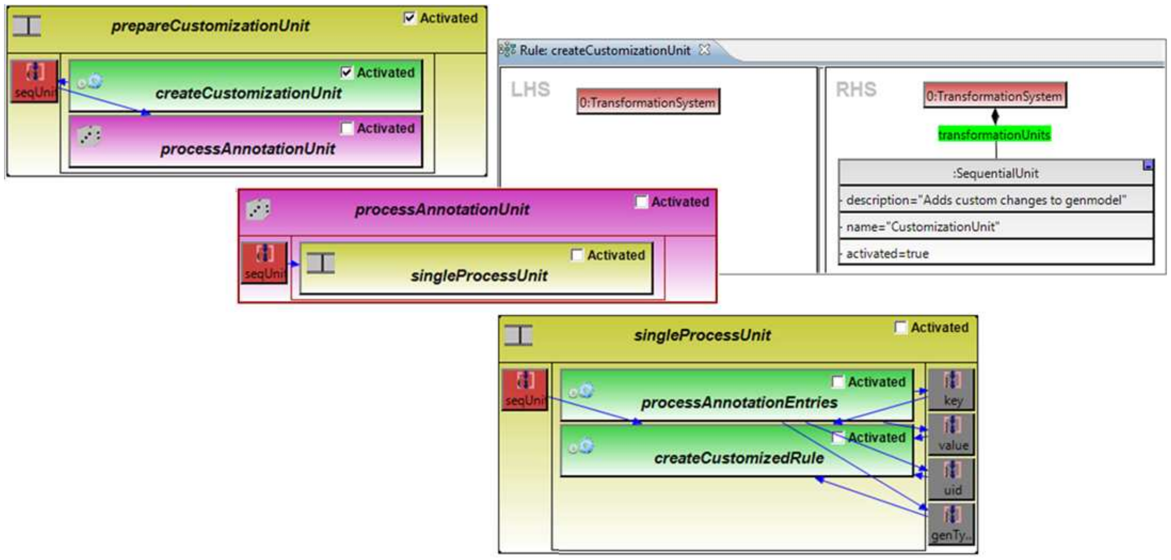


Fig. 3. Transformation units for processing annotated Ecore models

5 Conclusion

We presented a transformation from Ecore models to the GenModel format using the EMF transformation tool Henshin. Our solution is made available under SHARE via link `http://is.tm.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC10_Henshin.vdi`. We propose a solution for the basic case study and for Extension 2 considering also GenModel annotations in the source Ecore model and using Henshin's reflection ability to generate customized rules to set attributes of different `GenModel` classes. Being able with Henshin to work directly on EMF models and to define visual rules and control units helped a lot to come up with a straightforward translation algorithm.

References

1. TFS-Group, TU Berlin: EMF Tiger. (2009) `http://tfs.cs.tu-berlin.de/emftrans`.

2. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical definition of in-place transformations in the Eclipse Modeling Framework. In: Proc. MoDELS'06. Volume 4199 of LNCS. Springer, Berlin (2006) 425–439
3. Biermann, E., Ermel, C., Lambers, L., Prange, U., Taentzer, G.: Introduction to AGG and EMF Tiger by modeling a conference scheduling system. Software Tools for Technology Transfer (2010) To appear.
4. TFS-Group, TU Berlin: AGG. (2009) `http://tfs.cs.tu-berlin.de/agg`.
5. Biermann, E., Ermel, C., Taentzer, G.: Precise semantics of EMF model transformations by graph transformation. In: Proc. MoDELS'08. Volume 5301 of LNCS., Springer (2008) 53–67

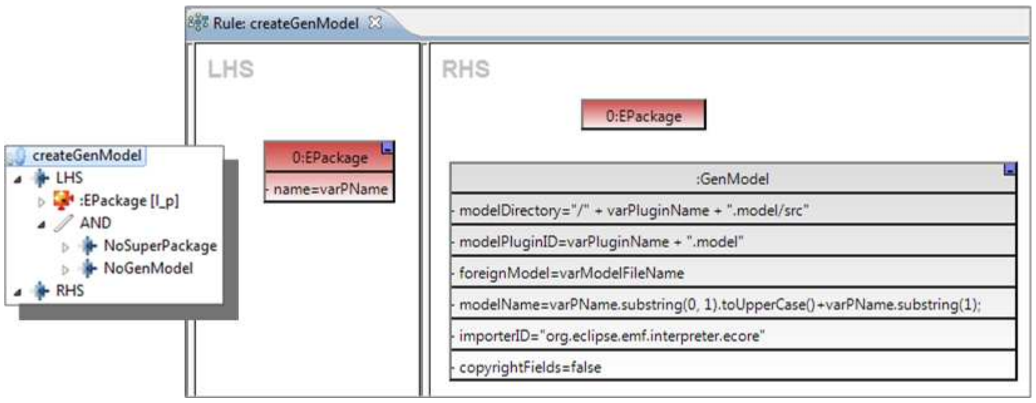
A Java Code of the Transformation Application

```

package tcc10;

import java.io.File;
import java.io.IOException;

import org.eclipse.emf.codegen.ecore.genmodel.GenModel;
import org.eclipse.emf.codegen.ecore.genmodel.GenModelPackage;
import org.eclipse.emf.codegen.ecore.genmodel.impl.GenModelPackageImpl;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EPackage;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
import org.eclipse.emf.ecore.xmi.impl.EcoreResourceFactoryImpl;
import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;
import org.eclipse.emf.henshin.common.util.EmfGraph;
import org.eclipse.emf.henshin.interpreter.EmfEngine;
import org.eclipse.emf.henshin.interpreter.UnitApplication;
import org.eclipse.emf.henshin.model.SequentialUnit;
import org.eclipse.emf.henshin.model.TransformationSystem;
import org.eclipse.emf.henshin.model.TransformationUnit;
import org.eclipse.emf.henshin.model.impl.HenshinPackageImpl;
import org.eclipse.emf.henshin.model.resource.HenshinResourceFactory;

/**
 * This implementation of an Ecore to Genmodel transformation by <a
 * href="http://www.eclipse.org/modeling/emft/henshin/">Henshin</a> was
   created
 * along the <a
 * href="http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/">Transformation
   Tool
 * Contest 2010</a> organized as satellite workshop to <a
 * href="http://malaga2010.lcc.uma.es/">TOOLS 2010</a>.<br>
 * Authors are (in alphabetical order):
 * <ul>
 * <li>Enrico Biermann
 * <li>Claudia Ermel
 * <li>Stefan Jurack
 * </ul>
 *
 * <i>Remark:</i> As proof of concept only, in the following source
     (.ecore) and
 * target (.gemodel) model files are hard-coded. However, an adaption to a
 * full-fledged plugin providing a context menu entry for ecore files is
 * straightforward.
 */
public class Ecore2GenmodelTrafo {

        /** Definition of a number of file paths */
        private static final String BASE = "model/";

        /** Mapping model */
        private static final String ECORE_E2G = "ecore2gen.ecore";
        private static final String ECORE_E2G_FULL = BASE + ECORE_E2G;
        /** Henshin file containing relevant rules */
        private static final String HENSHIN_E2G_FULL = BASE
                        + "Ecore2Genmodel.henshin";
        /** Ecore source model to be transformed */
        private static final String ECORE_SOURCE = "flowchartdsl.ecore";
        private static final String ECORE_SOURCE_FULL = BASE + ECORE_SOURCE;
        /** Genmodel target model */
        private static final String GENMODEL_TARGET_FULL = BASE
                        + "flowchartdsl2.genmodel";

        /** Common resource set */
        ResourceSet resourceSet = new ResourceSetImpl();

        /**
         * Method comprising the main control flow for the transformation.
         */
        public void generateEcore2Genmodel() {

                initializeResourceFactories();

                TransformationSystem ts = (TransformationSystem)
                    loadModel(HENSHIN_E2G_FULL);
                EPackage mappingModel = (EPackage)
                    loadModel(ECORE_E2G_FULL);

                EPackage ecoreModel = (EPackage)
                    loadModel(ECORE_SOURCE_FULL);

                // Create Henshin interpreter objects
                EmfGraph graphM = new EmfGraph();
                graphM.addRoot(ecoreModel);
                EmfEngine engineM = new EmfEngine(graphM);

                // Generate genmodel from ecore model (without annotations).
```

```
		TransformationUnit unit1 =
		    ts.findUnitByName("generateGenModel", true);
		UnitApplication unitApp1 = new UnitApplication(engineM,
		    unit1);
		// file name and plugin name cannot be reliably deduced by
		    the model
		// elements thus need to be set.
		unitApp1.setPortValue("inModelFileName", ECORE_SOURCE);
		unitApp1.setPortValue("inPluginName", ecoreModel.getName());
		boolean result = unitApp1.execute();

		graphM.addRoot(ts);
		graphM.addRoot(GenModelPackage.eINSTANCE);
		graphM.addRoot(mappingModel);

		// Process annotations and generate related Henshin rules.
		TransformationUnit unit2 = ts.findUnitByName(
				"prepareCustomizationUnit", true);
		UnitApplication unitApp2 = new UnitApplication(engineM,
		    unit2);
		unitApp2.execute();

		// Apply generated rules to transfer annotations to the
		    genmodel.
		SequentialUnit customizationUnit = (SequentialUnit) unitApp2
				.getPortValue("seqUnit");
		UnitApplication unitApp3 = new UnitApplication(engineM,
				customizationUnit);
		unitApp3.execute();

		// Save resulting genmodel.
		if (result) {
			System.out.println("Successful");
			GenModel gm = (GenModel)
			    unitApp1.getPortValue("outGenModel");
			saveGenModel(gm);
		} else {
			System.out.println("Not successful");
		}// if else

	}// generateEcore2Genmodel

	/**
	 * Saves the content of the genmodel to the specified file (see
	 * {@link #createGenModelResource()}).
	 *
	 * @param gen
	 */
	private void saveGenModel(GenModel gen) {
		URI modelUri = URI.createFileURI(new
		    File(GENMODEL_TARGET_FULL)
```

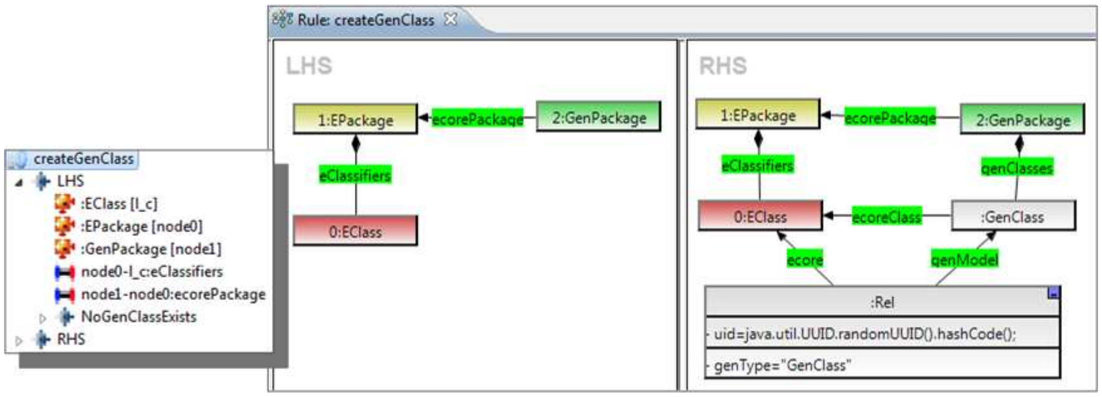
```
                                .getAbsolutePath());
                Resource res = resourceSet.createResource(modelUri,
                    "genmodel");
                try {
                        res.getContents().add(gen);
                        res.save(null);
                } catch (IOException e) {
                        e.printStackTrace();
                }// try catch
        }// saveGenModel

        /**
         * Loads the model at the given path and returns the root element.
         *
         * @param modelPath
         * @return
         */
        private EObject loadModel(String modelPath) {
                URI modelUri = URI.createFileURI(new
                    File(modelPath).getAbsolutePath());
                Resource resourceModel = resourceSet.getResource(modelUri,
                    true);
                return resourceModel.getContents().get(0);
        }// loadEmfModel

        /**
         * Registers appropriate resource factories for <b>ecore</b>,
         * <b>genmodel</b> and <b>henshin</b> files.
         */
        private void initializeResourceFactories() {
                Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
                                "ecore", new EcoreResourceFactoryImpl());
                Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
                                "genmodel", new XMIResourceFactoryImpl());
                Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
                                "henshin", new HenshinResourceFactory());

                // Initialize packages
                GenModelPackageImpl.init();
                HenshinPackageImpl.init();
        }// initializeResourceFactories

        /**
         * @param args
         */
        public static void main(String[] args) {
                Ecore2GenmodelTrafo s = new Ecore2GenmodelTrafo();
                s.generateEcore2Genmodel();
        }// main

}// class
```

B Rules contained in Unit `generateGenModel`



```
modelDirectory="/" + varPluginName + ".model/src"
modelPluginID=varPluginName + ".model"
foreignModel=varModelFileName
modelName=varPName.substring(0, 1).toUpperCase()+varPName.substring(1);
importerID="org.eclipse.emf.interpreter.ecore"
copyrightFields=false
```

Fig. 4. Rule `createGenModel`



```
uid=java.util.UUID.randomUUID().hashCode();
genType="GenClass"
```

Fig. 5. Rule `createGenClass`

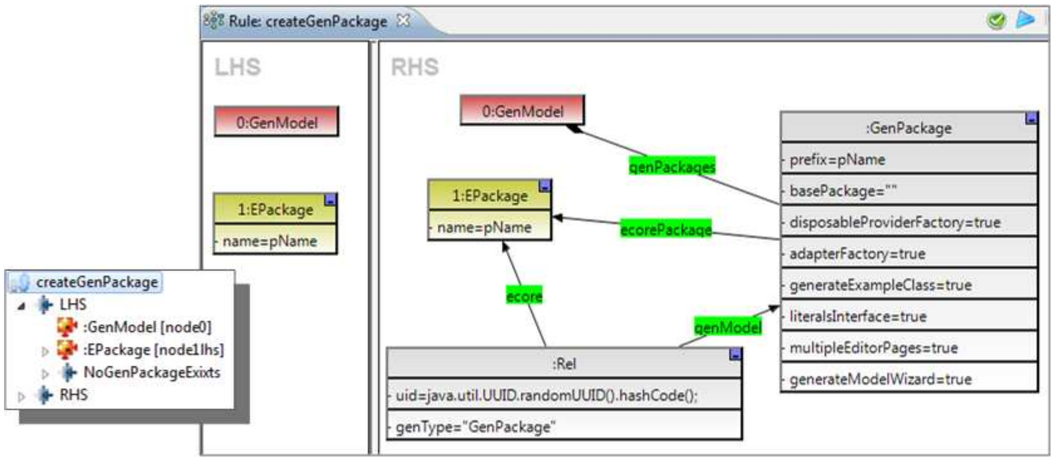


Fig. 6. Rule `createGenPackage`

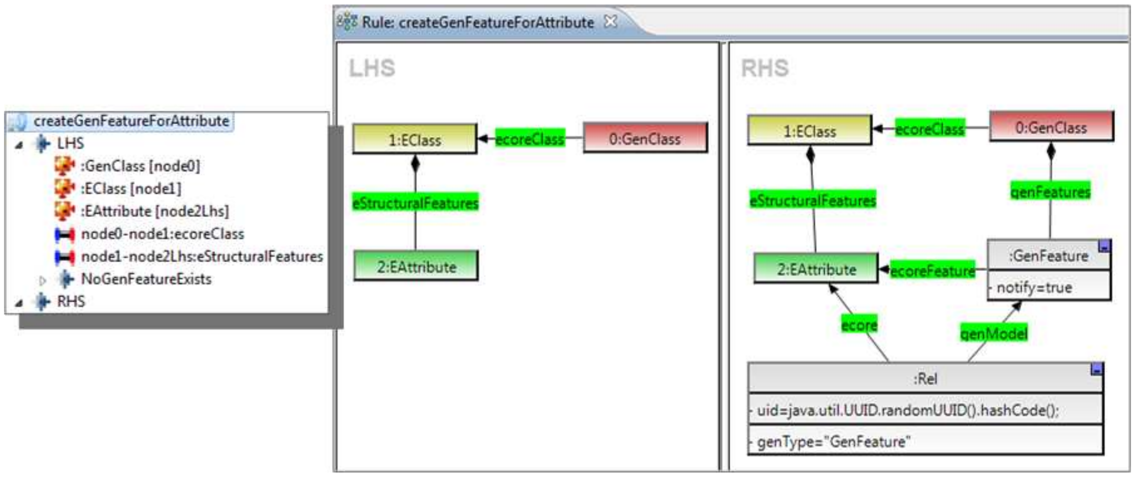


Fig. 7. Rule `createGenFeatureForAttribute`

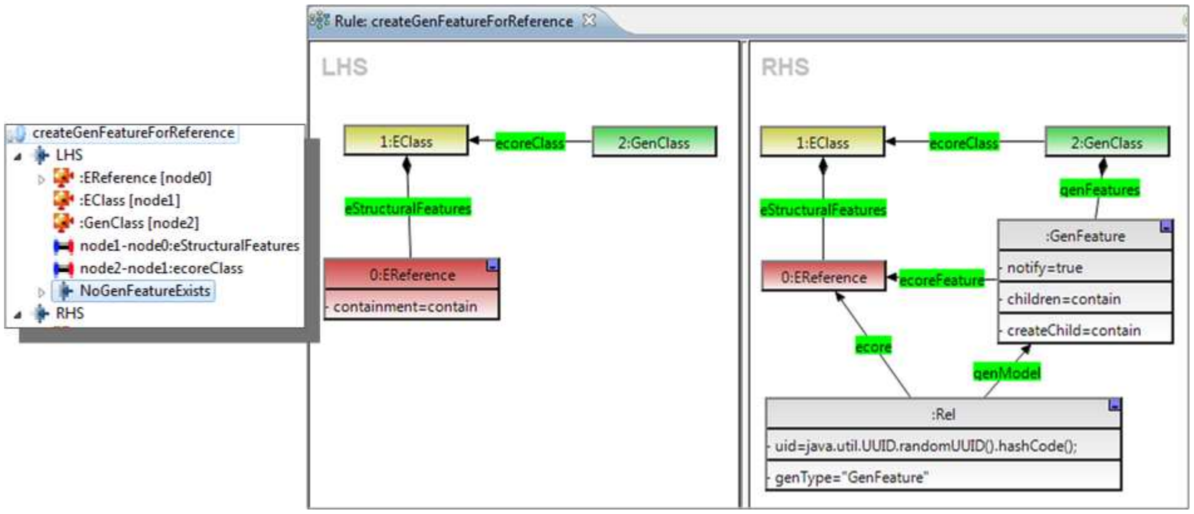


Fig. 8. Rule `createGenFeatureForReference`

C Rules contained in Unit `singleProcessUnit`

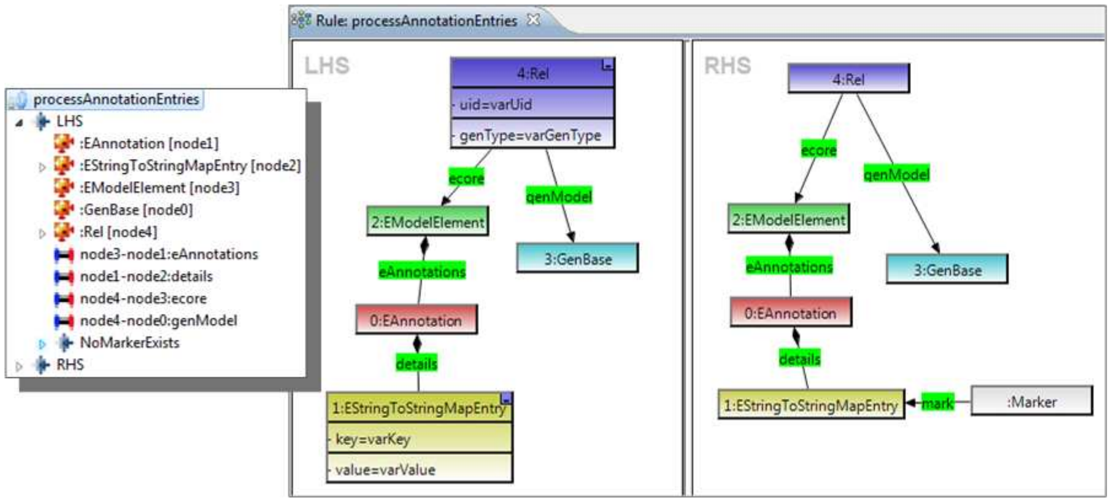


Fig. 9. Rule `processAnnotationEntries`

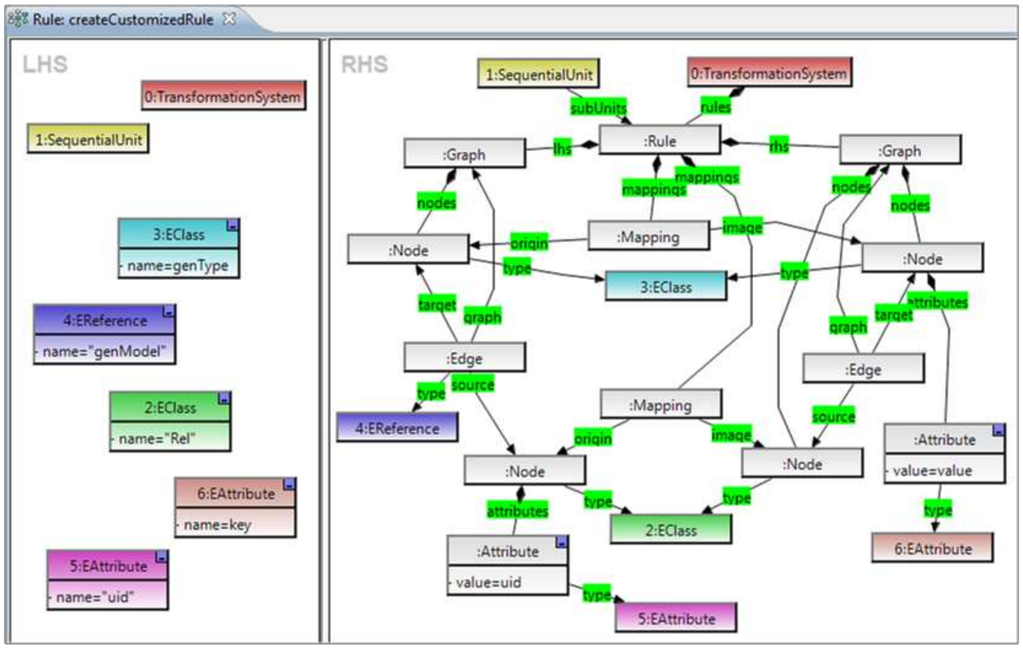
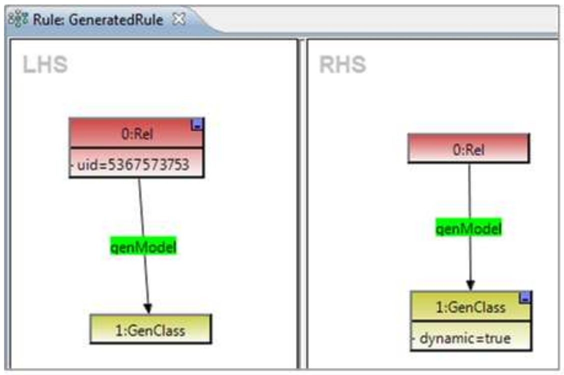


Fig. 10. Rule `createCustomizedRule`



generated by rule createCustomizedRule with the parameters:

uid = 5367573753
genType = “GenClass“
key = “dynamic“
value = true

Fig. 11. Rule `GeneratedRule`: Example for a generated rule