Tiger EMF Model Transformation Framework (EMT)

Version 1.2.0

User Manual

TU Berlin – EMT Project Team:

Enrico Biermann, Karsten Ehrig, Claudia Ermel, Christian Köhler, Günter Kuhns, Gabi Taentzer

Email: mailto:emftrans@cs.tu-berlin.de

Homepage: http://tfs.cs.tu-berlin.de/emftrans

Contents

1 Overview

The Eclipse Modeling Framework (EMF) provides a modeling and code generation framework for Eclipse applications based on structured data models. Although EMF provides basic operations for modifying EMF based models, it is still very difficult to define more complex operations on these models in a convenient way. The goal of the EMF Model Transformation framework (EMT) is to solve this issue by providing the means to graphically define rule-based transformations on EMF models. Our framework currently consists of three components:

- A graphical editor for visually defining EMF model transformation rules.
- A compiler, generating Java code from these rules that can be included into existing projects.
- An interpreter, allowing formal analysis of the rules using AGG, a graph transformation engine allowing formal analysis like termination and confluence of a graph transformation system.

All these components use a common model to describe transformation rules which are typed over EMF models.

1.1 Requirements

Before you can use these components you first have to fulfill the following software requirements:

- Java 5.0 Runtime Environment
- Eclipse 3.2
- EMF 2.2
- GEF 3.2

1.2 Installation

You can download and install EMT from the EMT project homepage `http://tfs.cs.tu-berlin.de/emt` or by using the update site `tfs.cs.tu-berlin.de/emt/update`. To include this update site in your Eclipse workspace follow these steps:

- Select Help→Software Updates→Find and Install
- Select Search for new features to install
- Choose `New remote site`
- Enter a name and as the URL enter `tfs.cs.tu-berlin.de/emt/update`
- Now make sure your new entry is selected in the list of update sites and press Finish.

2 Editor

2.1 Definition of EMF Model Transformations

Transformations for EMF models are defined using transformation rules. These rules consist of a left-hand side (LHS), a right-hand side (RHS), possible negative application conditions (NACs) and mappings between these so-called object structures. An object structure consists of a number of possibly linked objects typed over the EMF models for which the transformation is defined. Each of these structures is visualized in the graphical editor by a diagram that contains a number of object nodes, that can be connected and/or attributed.

The left-hand side of a rule stands for the structural preconditions that must be fulfilled to apply the rule. Accordingly a right-hand side describes the result (or postconditions) of a rule. Negative application conditions are defined in the same way and describe structural conditions that must not be fulfilled to apply the rule. Furthermore it is possible to define a layer for each rule. Rules on lower layers are applied prior to those on a higher layer.

Objects in the LHS of a rule can be mapped to objects in the RHS and also to objects in the NACs. The editor visualizes mappings by coloring the mapped objects in the same way or optionally displays corresponding numbers next to those objects. Those objects in the LHS, which are mapped to the RHS, will be preserved during rule application. Objects in the LHS, which have no mapping to the RHS are being removed. Accordingly, objects in the RHS without a mapping will be created during rule application.

An important property of EMF classes is the possibility of defining primitive-valued attributes. EMF Model Transformations usually involve calculations on these attributes, which can be defined directly in the graphical editor. Attributes of an object can be calculated using Java expressions, that must have the same type as the attribute defined in the EMF model. For these calculations, the expressions can contain variables which must be defined before they can be used.

2.2 Editor perspective

For the definition of EMF model transformations you can use the `EMF Model Transformation` perspective that is accessible once you install the EMT framework. You can change the current perspective via Window→Open perspective→Other. In Figure 1 you can see a small screenshot of this perspective showing a model transformation from activity diagrams to petrinets.

The perspective consists of five parts:

1. A tree-based view on the transformation in the upper left part
2. An editor to define properties of rules in the lower left part
3. A read-only view on the loaded ecore models (lower center)
4. A three-pane editor for defining rules (upper center)
5. A palette for selecting model objects and placing them in the rule (right part)

2.3 Working with the editor

Defining a new model transformation for EMF usually starts with the creation of an *.emt file. Creating a new *.emt file can be done via selecting EMF model transformation→EMF model transformation in the `New` wizard of Eclipse. In the upcoming wizard you must enter the project which should contain the *.emt file and its name. Now you can import a number of EMF models (*.ecore, *.xsd) using the "Import Packages"-Button in the Toolbar of Eclipse, see figure 2. You

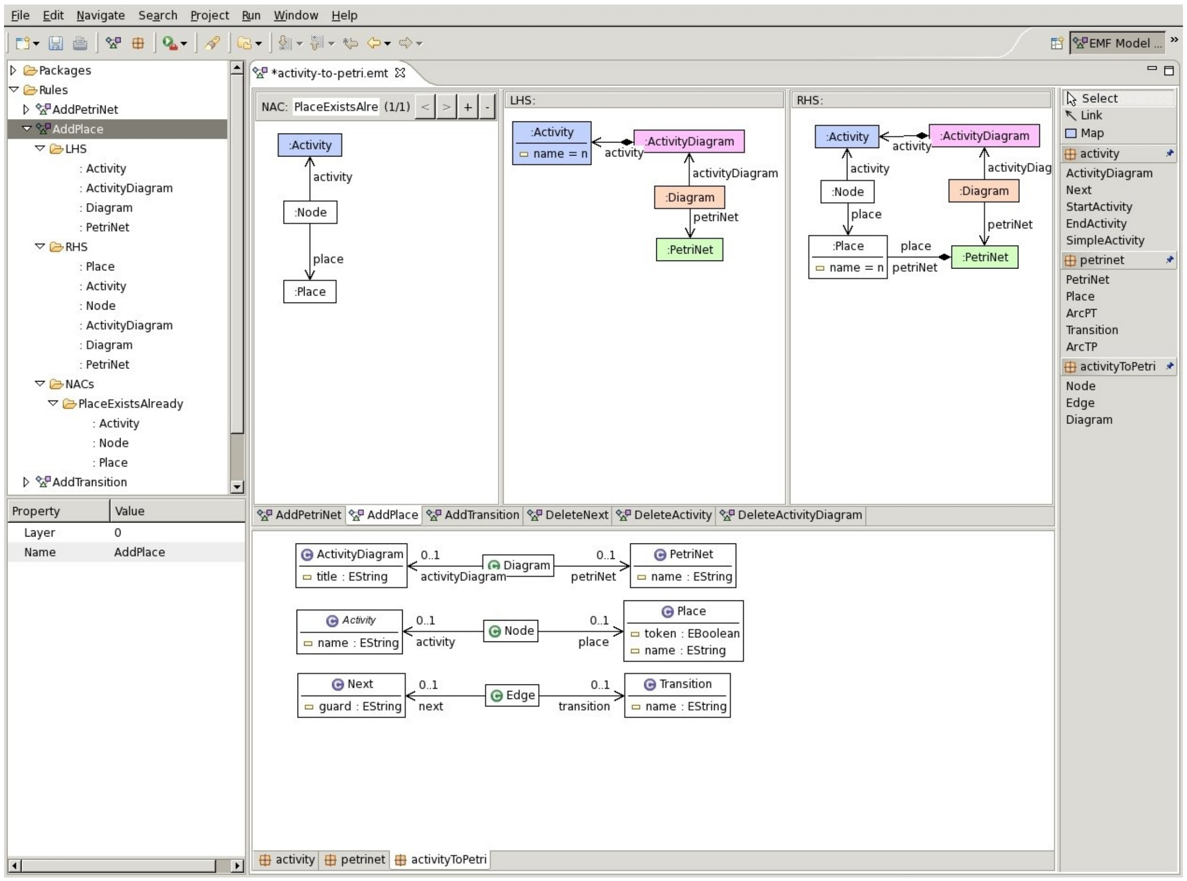


Figure 1: EMF model transformation perspective

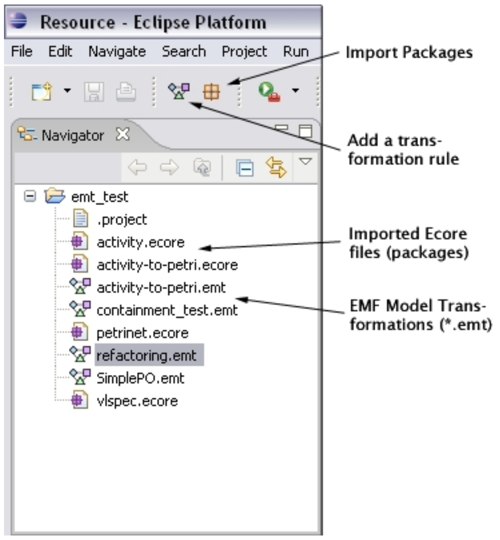
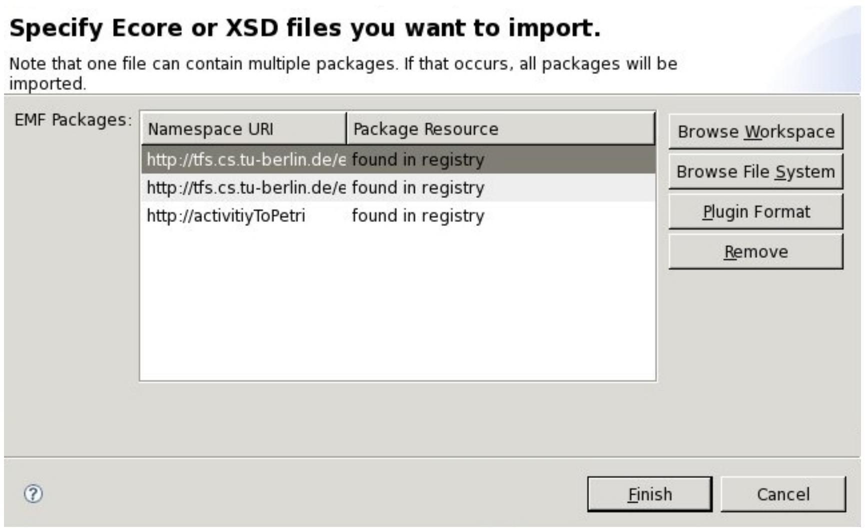


Figure 2: Eclipse integration

will see a wizard similar to Figure 3. You must provide either the path to the ecore file you want to use as a model or specify the model URI, for example `http://www.eclipse.org/emf/2002/Ecore`.

For each package, a diagram in the lower part of the editor is opened and entries in the palette are added on the right side and in the bottom part. In Figure 1 we have imported three models: activity,



Specify Ecore or XSD files you want to import.

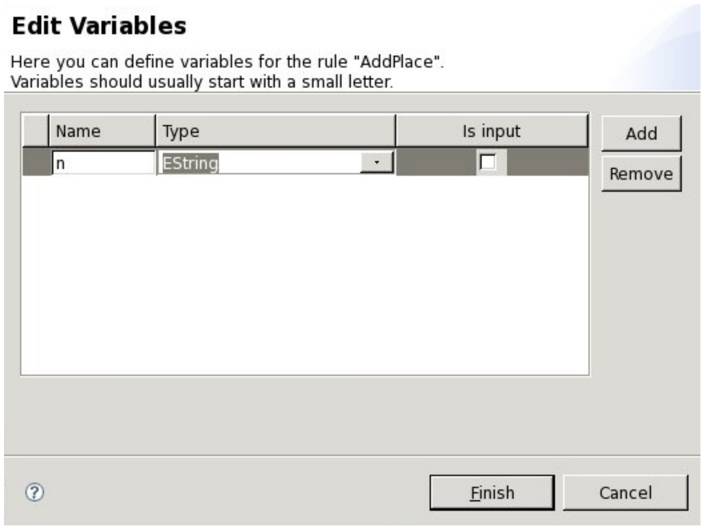
Note that one file can contain multiple packages. If that occurs, all packages will be imported.

EMF Packages:

Namespace URI	Package Resource
http://tfs.cs.tu-berlin.de/e	found in registry
http://tfs.cs.tu-berlin.de/e	found in registry
http://activitiyToPetri	found in registry

Figure 3: Import package wizard

petrinet and activityToPetri all from their corresponding ecore file. After you have imported at least one EMF model, you can start to define rules conforming to these models. To create a rule, click on the `Add Rule` button also shown in Figure 2. Now you can add objects to the diagrams using the palette entries. The objects can be connected or mapped to other diagrams using the "Link"-Tool or the "Map"-Tool in the palette, respectivly. Mappings are visualized through an equal coloring of the objects or by a preceding number which can be enabled from the context menu via View→Show mapping index. Attributes can be defined via the context menu of an object. Assigning Java expressions to the attributes can be either done in the properties view, or directly in the diagram by clicking on an already selected attribute. Variables used in expressions must be first declared using the "Edit variables" entry in the context menu of a diagram. When defining variables like seen in Figure 4 you can also choose whether the variable should be an input parameter or not. When you choose that a variable is an input parameter, you must explicitly



Edit Variables

Here you can define variables for the rule "AddPlace".
Variables should usually start with a small letter.

Name	Type	Is input
n	EString	☐

Figure 4: Variables dialog

pass a value for that parameter when applying a rule. Input parameter can be used for example to initialize attributes of a new class.

Advanced features, like consistency checks and customized views can all be configured using context menu entries. Testing the defined transformation rules is currently only possible by generating Java code or using the AGG based interpreter. A visual debugger is planned for the future.

3 Interpreter

The EMT interpreter performs EMF model transformations using the AGG engine. AGG provides an environment for creating and executing attributed graph transformations.
The interpreter converts an EMT grammar and an EMF model instance to AGG, executes the requested transformation steps using the AGG transformation engine and translates the result back to EMF.

3.1 Initialization

At first, the interpreter needs to be initialized with an EMT grammar which we defined by using the visual editor. To initialize it, we call the constructor of the interpreter:

```
Interpreter interpreter = new Interpreter(URI.createFileURI(file));
```

`file` is an absolute or relative path to the EMT grammar file where the transformation rules are stored. If, at some later point, you want to switch to an alternative EMT grammar, you can use the loadTransformation(URI) method:

```
interpreter.loadTransformation(URI.createFileURI(file));
```

After any of these steps, the interpreter is ready to be used.

3.2 Usage

For the actual transformation of a given EMF model instance, you can use one of the following two methods:

```
interpreter.transform(EObject root);
```

and

```
interpreter.applyRule(EObject root, String rulename, Vector mappings, Parameter parameter);
```

The parameter, `EObject root`, both methods have in common, is the topmost EObject in the containment hierarchy of the model instance that should be considered for matching purposes.

The method `transform` will apply all rules specified in the EMT file in a random order and as long as possible, starting with the EMF model instance specified by the root object. If you specified a layered grammar, it will honor the layer structure by not applying rules of a layer as long as a rule from the previous layer is applicable.

It is possible that the transform method may not terminate if certain rules are always applicable.

The `applyRule` method can be used to apply a certain rule with a specific match to a model. The parameter `rulename` specifies the name of the rule. Make sure you match the name used in the EMT file exactly or the rule won't be applied. The third parameter `mappings` contains EObjects from the model. While creating a rule using the EMT Editor, you choose a specific order for all EObjects in the LHS. The same order is used by the mappings vector. You can choose to pass `null` for certain EObjects to force AGG to complete the match on its own. Consider for example the rule `AddTransition` in Figure 5, which converts a Next relation from an activity diagram into a transition of the corresponding petrinet. The NAC ensures that this rule is only applied in the case

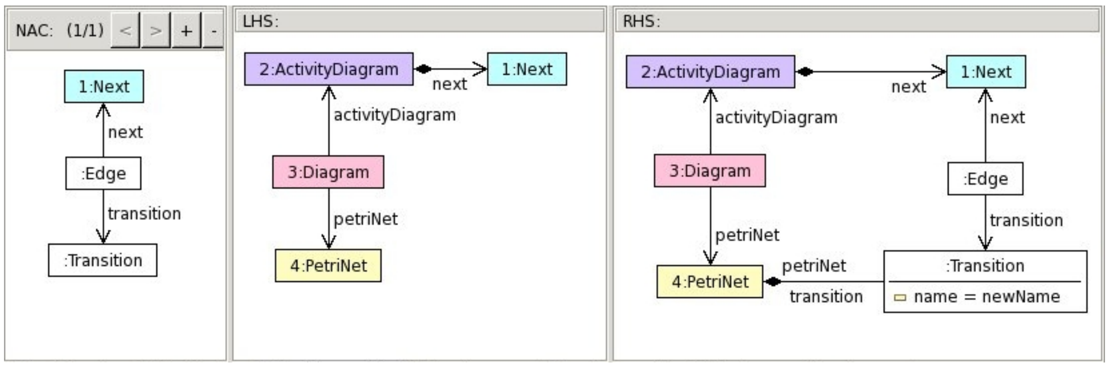


Figure 5: The rule AddTransition

that the Next relation has not yet been converted. The LHS of this rule consists of four different objects. If you want to match the diagram and let the rest of the match be completed automatically you would enter the following code:

```
Vector mappings = new Vector();
mappings.add(null);
mappings.add(null);
mappings.add(fixedDiagramEObject);
mappings.add(null);
```

In this case AGG will try to find a match for the `Next`, `ActivityDiagram` and `PetriNet` objects while keeping the `Diagram` fixed. You can also pass `null` for the whole vector so a random match will be chosen. The last `applyRule` parameter `Parameter` is used to set the input parameters for the rule as defined in the EMT file. Input parameter can be defined in the following way:

```
Parameter parameter = new Parameter();
parameter.addParameter(String name, EObject value, String type);
```

In case of primitive types, for example:

```
parameter.addIntParameter(String name, int value);
```

A valid code block for defining the input parameter for the rule `AddTransition` would look like this:

```
Parameter parameter = new Parameter();
parameter.add("newName", "transition1", "String");
```

After defining the match and the input parameters, the rule can be applied by:

```
interpreter.applyRule(root, "AddTransition", mapping, parameter);
```

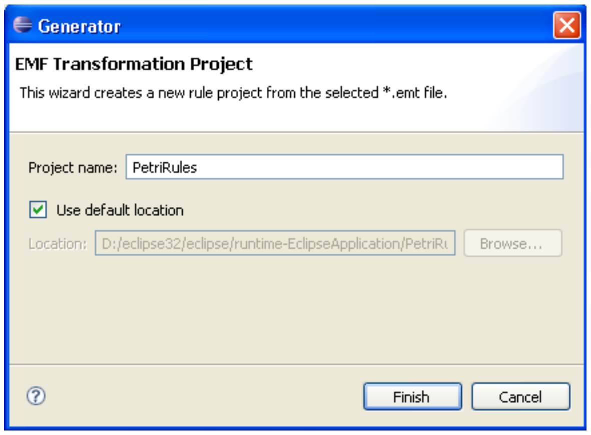
4 Compiler

The Compiler uses the EMT grammar file to create java classes for all rules defined within the file. So using the compiler usually consists of two parts:

1. Compiling the EMT grammar file to Java code
2. Using the Java classes to perform model transformations.

4.1 Compiling

To start the compiler you can use the context menu in Eclipse of either an EMT grammar file or anywhere in the graphical editor. Choose the entry `Generate rule classes`. You will see a dialog similar to Fig. 6 where you can choose the name of the rule project. After clicking on finish, the



This wizard creates a new rule project from the selected *.emt file.

Project name: PetriRules

☑ Use default location

Location: D:/eclipse32/eclipse/runtime-EclipseApplication/PetriRu

Figure 6: dialog for the Compiler

code generation will start and a new project will be created which contains all classes needed to perform model transformations as defined in the EMT grammar file.

4.2 Usage

To describe how to use the generated code consider the rule *AddTransition* shown in Fig. 5 we already used with the interpreter.

In the project generated by the compiler you can find a class for this rule which starts with the rule name and finishes with "Rule", in this case the class will be called `AddTransitionRule`. This class can be integrated into existing code to transform model instances in the way described by the rule. To execute that rule an instance of this class must be created first:

```
AddTransitionRule atr = new AddTransitionRule(root);
```

The constructor requires an *EObject* from a model instance, which is used as a root node - the rule is only applied to this node and all its child nodes. In this case we assume `root` is an arbitrary class that contains at least `ActivityDiagram`, `PetriNet` and `Diagram`. Before a rule is executed input parameters and mappings can be set:

```
//set parameters
cpr.setParNewName("transition1");
```

```
//set match for LHS
Vector match = new Vector();
match.add(null);
match.add(null);
match.add(fixedDiagram);
match.add(null);
cpr.setLHS(match);
```

Each rule class has a method `execute()` to apply the corresponding rule to a model instance. The return value of this method indicates if an application of this rule (with the given parameters and match) was successful or not. A successful application can be undone by `undo()` and after this redone by `redo()`. Th rule `atr` for example is executed by:

```
atr.execute();
```

The class `Transformation` offers an interface to all rule classes. Rules can be executed similar to the Interpreter by `applyRule(name, match, parameters)`. This is equivalent to calling `execute()` of that rule. Additionally the method `transform()` allows to apply rules in an arbitrary order to the instance, until no rule is applicable anymore. To establish some kind of control-flow on the rules, they can be grouped into layers in the editor which are applied sequentially like separate rule sets by this method.

Rule application changes the model instance in-place such that all references to unchanged parts are kept, since *EObjects* are preserved. Deleted *EObjects* are saved and can be restored if a rule application is undone. If you execute more than one rule, you can completely undo the transformation sequence by calling the undo method of all created rule instances in a reverse order. A redo function is also avaible.

5 Example: Refactoring the PetriNet Model

The goal of our example is to refactor the petrinet model in figure 7 such that the attribute `name` of the classes `Transition`, `Place` and `PetriNet` is moved to a new superclass called `NamedElement` as seen in figure 8. Please note that in this case we are not transforming a simple instance of a model but a model conforming to a metamodel. This is possible because the Ecore metamodel can be defined in its own language, so in our example we view the Ecore metamodel as the model and the Petrinet model as an instance of this model.

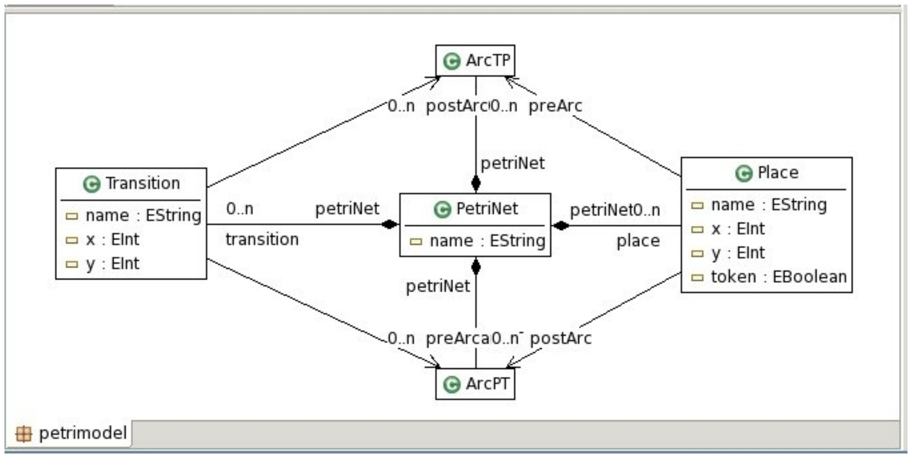


Figure 7: Old petrinet model

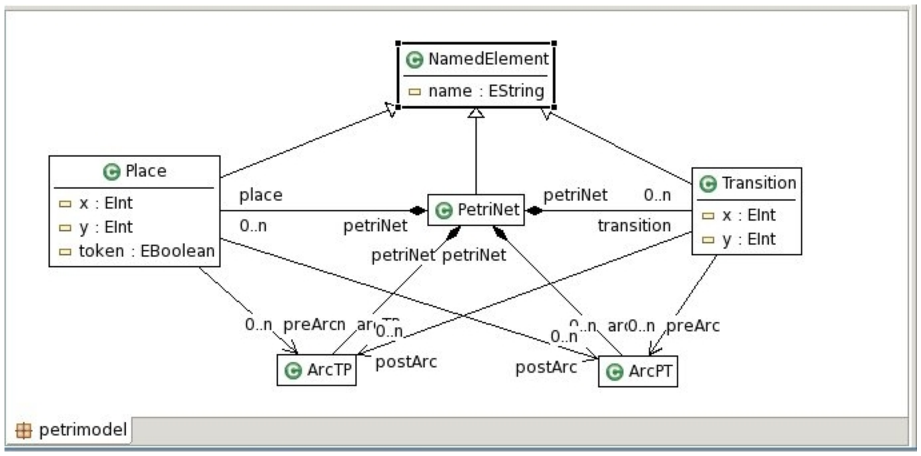


Figure 8: New petrinet model

5.1 EMT Grammar

In the de.tuberlin.emt.examples project you can find the file src/Ecore/emt/refactoring.emt. This file contains all rules needed to perform the refactoring operations. The rules we need to perform the necessary refactoring steps can be seen in Fig. 9 - 14. `CreateSuperclass` for example creates a new superclass with name `s` and abstract state `a` for the class with the name `c`. `a`, `c` and `s` are input parameter in this rule. `ConnectSuperclass` can be used to set a subclass-superclass relation between two existing classes. It doesn't have any input parameter so you have to specify the two classes in the mapping. Finally the four rules `CheckAttribute`, `PullUpAttribute`, `DeleteAttribute` and `DeleteAnnotation` are used to perform the actual refactoring step:

- CheckAttribute checks if the passed attribute exists in all subclasses, see Fig. 11
- PullUpAttribute adds the attribute to the superclass and deletes it from one subclass, see Fig. 12
- DeleteAttribute deletes the attribute from all other subclasses, see Fig. 13
- DeleteAnnotation deletes a possible marker which would have been placed if the attribute couldn't be pulled up, see Fig. 14

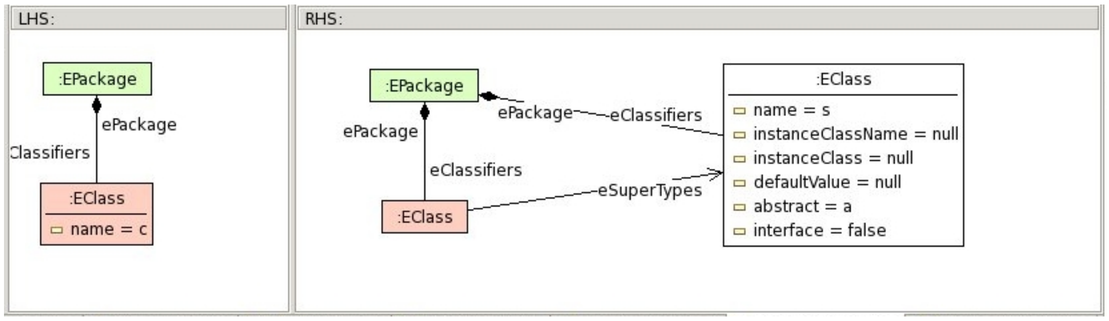


Figure 9: CreateSuperclassRule

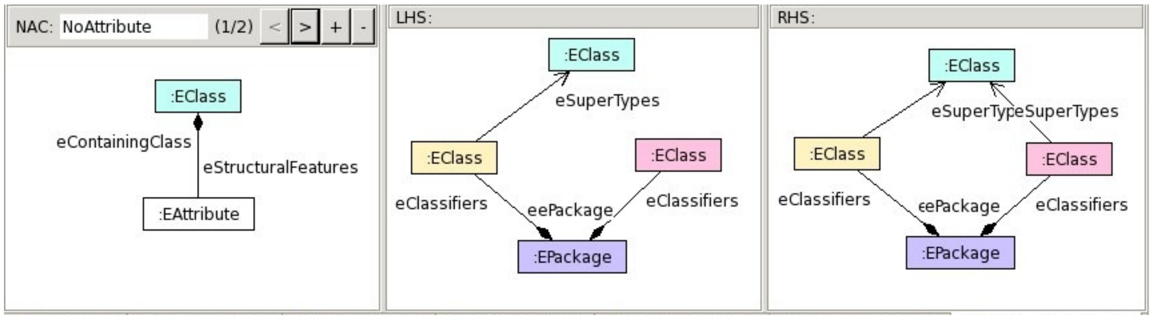


Figure 10: ConnectSuperclassRule

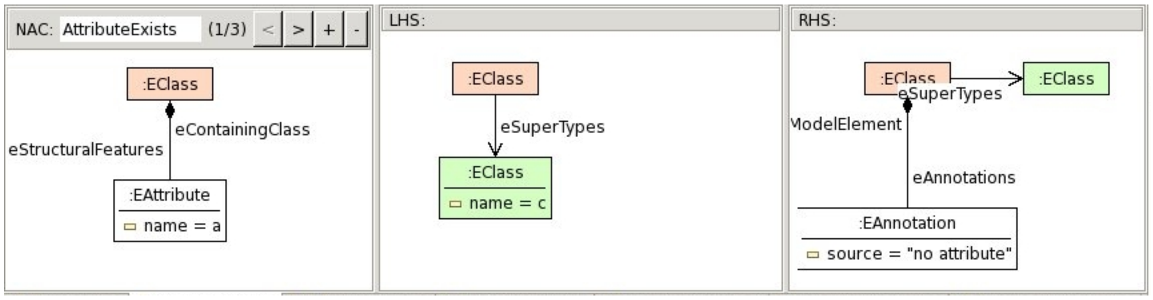


Figure 11: CheckAttributeRule

5.2 Import/Export of model instances

A common way to store EMF model instances is in the form of XMI files featuring an XML style syntax. Before you can load a model instance from such a file it might be necessary to register the specific file extension first. In this example we want to be able to import *.ecore files since those are the instances of the ecore model. To register the ecore extension you have to call:

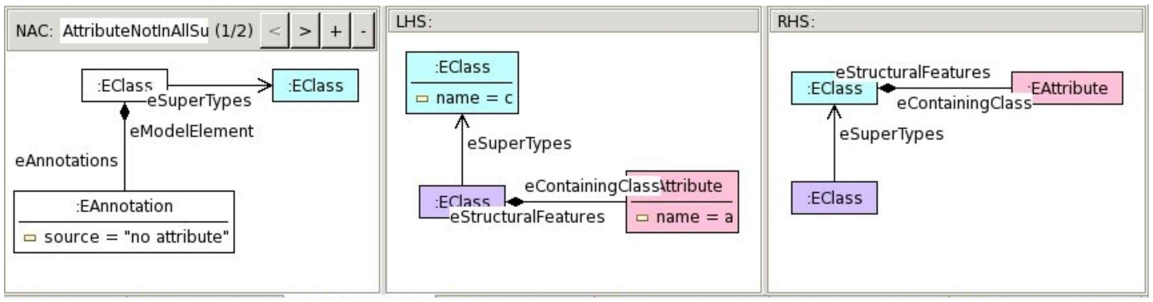


Figure 12: PullUpAttributeRule

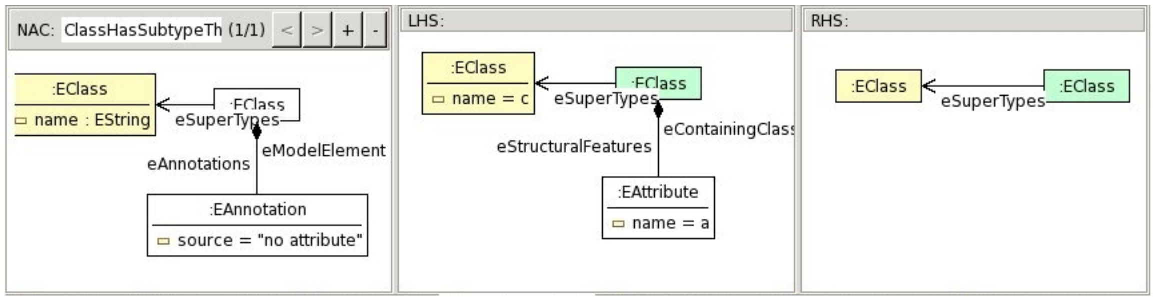


Figure 13: DeleteAttributeRule

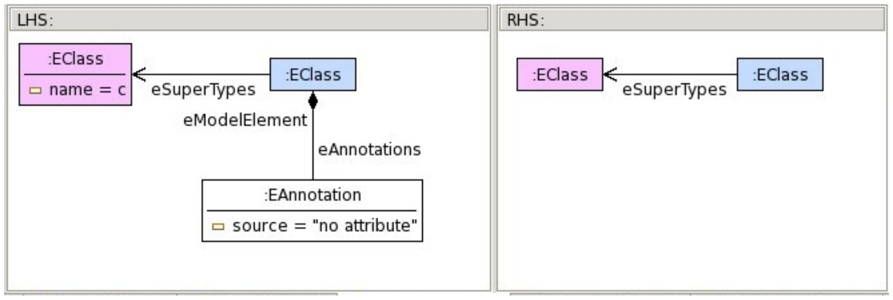


Figure 14: DeleteAnnotation

```
Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
"ecore", new XMIResourceFactoryImpl());
```

Make sure the org.eclipse.emf.ecore.xmi project is in the list of required plugins.

Now that the ecore files are known to the registry we can import them by calling:

```
ResourceSet resourceSet = new ResourceSetImpl();
Resource resource = resourceSet.getResource(
URI.createFileURI("src/Ecore/model/petrimodel.ecore"), true);
EPackage model = (EPackage) resource.getContents().get(0);
```

This code block will load the petrimodel.ecore from the specific location and allow access to the first object in this file via the variable `model`.

If at any point you want to store a certain model to a file, you can do this by calling:

```
Resource res = new XMLResourceImpl(
URI.createFileURI("src/model/petrimodel_refactored.ecore"));
```

```
res.getContents().add(model);
try {
res.save(null);
} catch (IOException e) {
}
```

5.3 Interpreter usage

First you have to create a new class either in an existing plugin project or a new one. Note that you can use the class InsertSuperclassInterpreter in the de.tuberlin.emt.examples project as a reference for using the interpreter.

Make sure that your project includes the packages:

1. de.tuberlin.emt.interpreter
2. de.tuberlin.emt.common
3. org.eclipse.emf.ecore.xmi (needed to load an EMF model from an XMI file)

in the list of required plugins.

Now that the project itself is prepared, load the petrinet model from src/Ecore/model/petrimodel.ecore as shown in Section 5.2.

Next you have to initialize an interpreter instance which you can use later to modify the model.

```
Interpreter interpreter = new Interpreter(URI.createFileURI("src/Ecore/emt/refactoring.emt"));
```

At this point you can start to apply rules to the model. For our example you might want to create a new superclass called NamedElement:

```
// create the new superclass
Parameter parameter = new Parameter();
parameter.addBooleanParameter("a", true);
parameter.addParameter("c", "PetriNet", "String");
parameter.addParameter("s", "NamedElement", "String");
interpreter.applyRule(model, "CreateSuperclass", null, parameter);
```

The specified parameter `a` specifies whether the created class will be abstract, `c` is the name of the class which will get a new parent and `s` is the name of the new superclass. Now you can connect Transition and Place to the new superclass:

```
// connect transition to new superclass
Vector<EObject> mapping = new Vector<EObject>();
mapping.add(null);
mapping.add((EClass)model.getEClassifier("Transition"));
interpreter.applyRule(model, "ConnectSuperclass", mapping, null);

// connect place to new superclass
mapping = new Vector<EObject>();
mapping.add(null);
mapping.add((EClass)model.getEClassifier("Place"));
interpreter.applyRule(model, "ConnectSuperclass", mapping, null);
```

At this point Place, Transition and PetriNet have a common superclass NamedElement without attributes. All three classes have the common attribute `name` we would like to move to the superclass. For this purpose we need the four rules: `CheckAttribute`, `PullUpAttribute`, `DeleteAttribute` and `DeleteAnnotation` seen in Fig. 11-14.

Since all four rules are needed for a refactoring step, we combine them in a method called refactoring. The input parameter `a` and `c` denote the name of the attribute and the name of the class the attribute should be pulled to.

```
public void refactoring(EPackage model, String attributeName, String targetClass) {
Parameter parameter = new Parameter();
parameter.addParameter("a", attributeName, "String");
parameter.addParameter("c", targetClass, "String");

interpreter.applyRule(model, "CheckAttribute", null, parameter);

do{}
while (interpreter.applyRule(model, "PullUpAttribute", null, parameter));

do{}
while (interpreter.applyRule(model, "DeleteAttribute", null, parameter));

do{}
while (interpreter.applyRule(model, "DeleteAnnotation", null, parameter));
}
```

Now all we have to do is call

```
refactoring(model, "name", "NamedElement");
```

and the attribute will be moved to NamedElement. Now you can store the refactored petrinet model as shown in section 5.2.

5.4 Compiler usage

Before you can use code for EMF model transformations, you must first generate the code from an EMT grammar. You evoke the compiler by right-clicking the refactoring.emt file and choose EMF Transformation→Generate rule classes. As the name for the new project enter RefactoringRules.

After the new project has been generated create a new plugin project. In the new project add the following projects to the list of required plugins:

1. RefactoringRules
2. org.eclipse.emf.ecore.xmi (needed to load an EMF model from an XMI file)

Copy the petrimodel.ecore file from the example project to src/model/petrimodel.ecore in your new project. Now create a new class with a main method. Register and import the petrimodel.ecore file into your class. At this point you can start to apply rules to the petrinet model. In the beginning you start by creating the new superclass NamedElement:

```
// create the new superclass
CreateSuperclassRule createSC = new CreateSuperclassRule(model);
createSC.setParA(true);
createSC.setParC("PetriNet");
createSC.setParS("NamedElement");
createSC.execute();
```

Parameter `a` means whether the new class is abstract or not, `c` is the name of the class which should get a new parent and `s` is the name of the superclass itself.
Next we need to connect Transition and Place to the new class using the class `ConnectSuperclassRule`:

```
// connect transition to new superclass
ConnectSuperclassRule connectSC1 = new ConnectSuperclassRule(model);
connectSC1.setEclass0((EClass)model.getEClassifier("Transition"));
connectSC1.execute();
// connect place to new superclass
ConnectSuperclassRule connectSC2 = new ConnectSuperclassRule(model);
connectSC2.setEclass0((EClass)model.getEClassifier("Place"));
connectSC2.execute();
```

In this case we didn't use a parameter to narrow down the matching process, but did specify the object itself. Now the classes Place, Transition and PetriNet have a new common superclass. Since all three classes have an attribute name we can move it to NamedElement. To do this we write a refactoring method that calls different rules to refactor a specific attribute:

```
public static void refactoring(EPackage model, String name, String targetClass) {
CheckAttributeRule checkAttribute = new CheckAttributeRule(model);
checkAttribute.setParA(name);
checkAttribute.setParC(targetClass);
checkAttribute.execute();

PullUpAttributeRule pullUp = null;
do {
pullUp = new PullUpAttributeRule(model);
pullUp.setParA(name);
pullUp.setParC(targetClass);
pullUp.setParT(type);
}
while(pullUp.execute());

DeleteAttributeRule deleteAttribute = null;
do {
deleteAttribute = new DeleteAttributeRule(model);
deleteAttribute.setParA(name);
deleteAttribute.setParC(targetClass);
deleteAttribute.setParT(type);
}
while(deleteAttribute.execute());

DeleteAnnotationRule deleteAnnotation = null;
do {
deleteAnnotation = new DeleteAnnotationRule(model);
deleteAnnotation.setParC(targetClass);
}
while(deleteAnnotation.execute());
}
```

In the main method enter the following line:

```
// pulls up the attribute "name"
refactoring(model, "name", "NamedElement");
```

Finally we want to save the newly refactored model to a file:

```
// save the new model to a file
Resource res = new XMLResourceImpl(URI.createFileURI("src/model/petrimodel_refactored.ecore"));
res.getContents().add(model);
try {
res.save(null);
} catch (IOException e) {
}
```

Now you have a new *.ecore file in your filesystem which contains the refactored petrinet model. You can view this file with the tree based EMF editor, the Omondo plugin (`www.omondo.com/`) or the example Ecore editor generated by GMF (`http://www.eclipse.org/gmf/`)

6 Example: From Class Diagrams To Relational Data Base Models

The example in this section is an exogenous model transformation from class models to relational data base models (*CD2RDBMS* for short). This quasi-standard model transformation has been originally defined in the specification for QVT [BRST05, QVT05] by the Object Management Group. Here, we present a restricted variant with attributes of primitive data types only. The model for the source language (shown in Fig. 15 (a)) consists of simple class diagrams. Please note that we spell *Klass* with a "*K*" since the compiler forbids the use of Java keywords like *class* in the models. The target language model (see Fig. 15 (b)) consists of schemes for database tables. A reference structure model is needed for exogenous model transformations establishing a helper structure for the model transformation. The model for the *CD2RDBMS* reference structure (Fig. 15 (c)) relates classes with tables and attributes with columns. Associations are related to foreign keys.

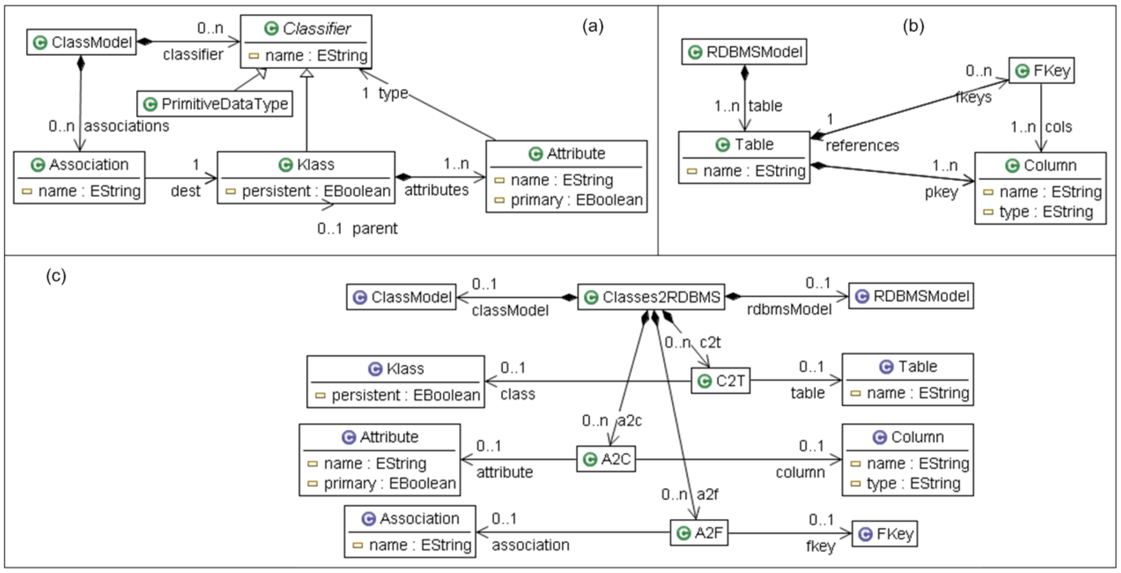


Figure 15: Source, Target and Reference Models for the*CD2RDBMS* Model Transformation

In the following, we explain the *CD2RDBMS* transformation rules. Note that we need to include the root container *Classes2RDBMS* in the LHS of a transformation rule if new objects are added to this container in the RHS of the rule. Fig. 16 shows two rules to convert classes to tables.

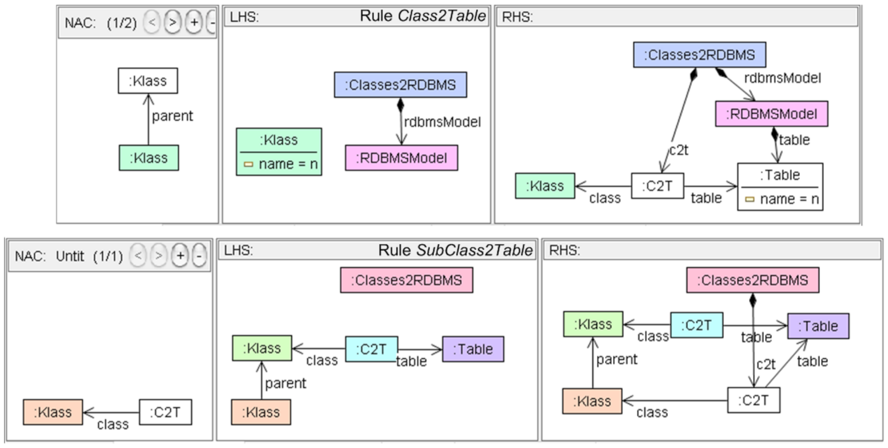


Figure 16: Rules *Class2Table* and *SubClass2Table*

Rule *Class2Table* creates a *Table* for each class which is not a subclass of another class. Rule *SubClass2Table* is applicable only to classes which have a superclass. The rule creates a relation between the class and the already existing table corresponding to its superclass.

Fig. 17 shows two rules to convert a class attribute to a columns of the table corresponding to the class. Rule *Attribute2Column* generates a column in the table of the class and sets the column name and type to the same values as the attribute's name and type. Rule *SetKey* sets a *pkey* relation between the table and the column if the corresponding class attribute was a *primary* attribute (*primary=true*).

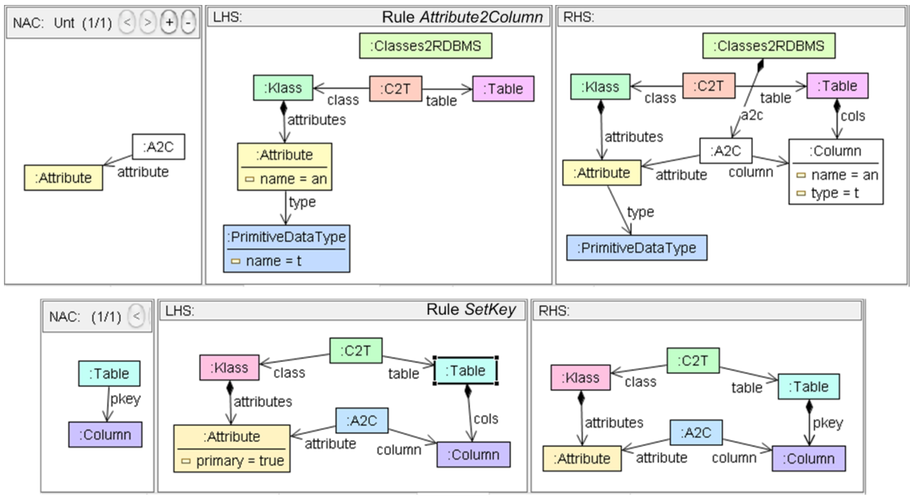


Figure 17: Rules *Attribute2Column* and *SetKey*

At last, associations are converted to foreign keys by rule Association2FKey shown in Fig. 18. In the table corresponding to the source class of the association, a foreign key is inserted which links a newly generated column to the table corresponding to the target class of the association. The new column gets a name composed of the association name and the name of the primary attribute of the target class.

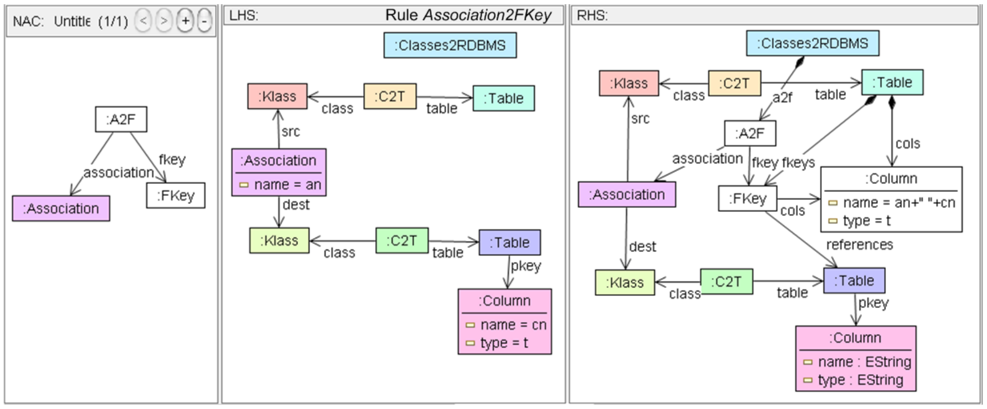


Figure 18: Rule *Association2FKey*

For this example, the compiler and interpreter usage works as described for the Petri net refactoring example in Section 5. We here give the code of a sample test class (Test.java) which is uses the compiled code to build up a model instance and calls the *transform* method to evoke the model transformation.

First, three factories for the source, target and reference model are defined. Using the factories, three models `cModel`, `rModel` and `c2r` are created. The three models are combined by setting the

`ClassModel` of the reference model c2r to `cModel`, and the `RdbmsModel` to `rModel`.

```
// define factories
Classes2rdbmsFactory c2rFactory = new Classes2rdbmsFactoryImpl();
ClassesFactory cFactory = new ClassesFactoryImpl();
RdbmsFactory rFactory = new RdbmsFactoryImpl();

// create models
Classes2RDBMS c2r = c2rFactory.createClasses2RDBMS();
ClassModel cModel = cFactory.createClassModel();
RDBMSModel rModel = rFactory.createRDBMSModel();

// combine models via reference model
c2r.setClassModel(cModel);
c2r.setRdbmsModel(rModel);
```

Now, the objects and links of a model instance can be defined. Fig. 19 shows the code of a model instance, and its graphical visualization.

```
Klass klass1 = cFactory.createKlass();
Klass klass2 = cFactory.createKlass();
Attribute attribute2 = cFactory.createAttribute();
PrimitiveDataType
    dataTypeInt = cFactory.createPrimitiveDataType();
Association ass1 = cFactory.createAssociation();

ass1.setName("test");
ass1.setSrc(klass1);
ass1.setDest(klass2);
klass1.setName("Klass1");
klass1.setPersistent(true);
klass2.setName("Klass2");
klass2.setPersistent(true);
attribute2.setName("Attribute2");
attribute2.setPrimary(true);
attribute2.setType(dataTypeInt);
dataTypeInt.setName("int");

cModel.getClassifier().add(klass1);
cModel.getClassifier().add(klass2);
cModel.getAssociations().add(ass1);
klass2.getAttributes().add(attribute2);
cModel.getClassifier().add(dataTypeInt);
```

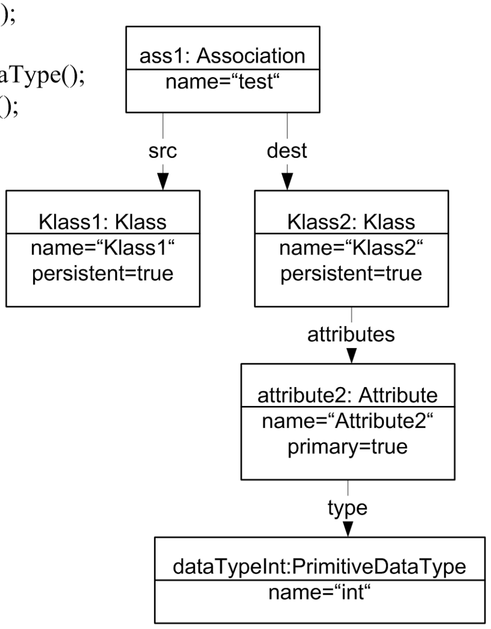


Figure 19: A model instance of the combined *CD2RDMS* model

After the model has been defined, the model transformation may be tested by defining a Transformation instance, and calling the transform() method on this instance.

```
Transformation test = new Transformation(c2r);
test.transform();
```

The result of the complete model transformation transforming the model instance in Fig. 19 is shown in Fig. 20. Note that we did not yet define rules to delete the class diagram objects and the reference objects, hence the result graph contains objects from all three (source, target and reference) models.

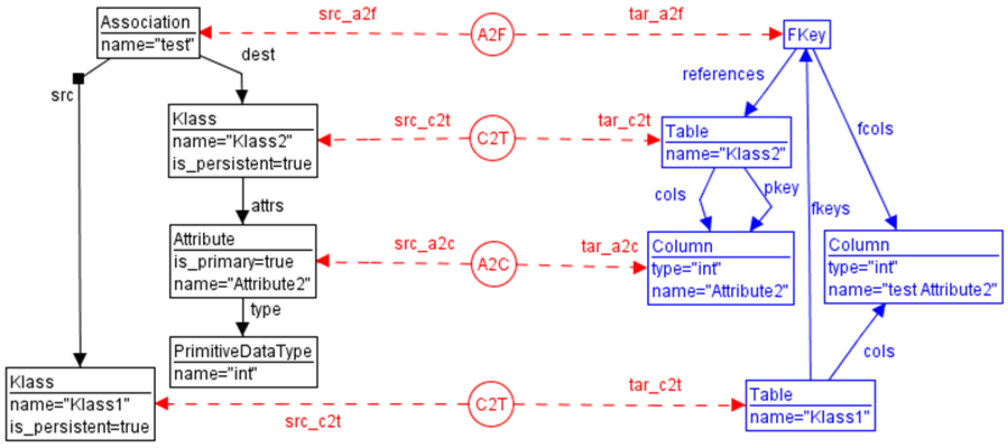


Figure 20: Model Transformation Result for the Class Diagram in Fig. 19

Single rule applications may also be tested, e.g. by the following code:

```
Class2TableRule c2t = new Class2TableRule(c2r);
c2t.execute();
```

The result of a transformation may be saved and viewed by the tree view editor which can be generated by EMF. Alternatively, the Eclipse debugger view allows to explore the resulting bindings for the transformed model instance.

7 Restrictions

- Up to now, the compiler generates code in a project named by the user. If the defined project name already exists (i.e. a project of the same name has been generated previously), the existing code will not be overwritten, i.e. nothing will be generated, and the user is prompted for a new project name. We envisage to implement a *Merger* like in Eclipse EMF, such that parts of the generated code can be overwritten (those parts marked by a `generated`-tag), and other parts (without `generated`-tag) remain unchanged by a new generation process. This enables the user to change parts of the generated code by hand, such that these parts will not be overwritten by each new generation process.
- In principle, references may be defined to objects that cannot be transitively reached from the root object via the containment hierarchy. Such objects are not considered by the match-finder for transformation purposes and should not be used in rules.

References

[BRST05] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model transformations in practice workshop. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 120–127. Springer, 2005.

[QVT05] *Query/View/Transformation (QVT). QVT-Merge Group, version 2.0 (2005-03-02). http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf*, 2005.