Solving the TTC 2011 Reengineering Case with HENSHIN

Stefan Jurack  
Universität Marburg, Germany  
sjurack@mathematik.uni-marburg.de

Johannes Tietje  
Technische Hochschule Mittelhessen, Gießen, Germany  
johannes.tietje@mni.th-mittelhessen.de

This paper presents the HENSHIN solution to the Model Transformations for Program Understanding case study as part of the Transformation Tool Contest 2011.

1 Introduction

Models are a helpful means of representing different aspects of a software system more abstractly to improve comprehension. In the modeling community, the Eclipse Modeling Framework (EMF) has evolved to a widely used technology. While EMF itself provides modeling and code generation capabilities, extensions such as the Java Model Parser and Printer (JaMoPP) allow the translation of Java source code into equivalent EMF model representations. This paves the way to exploit model-to-model transformations in order to translate source code models into other possibly more abstract representations.

HENSHIN is a declarative transformation language and tool environment for in-place EMF model transformation. In-place means that EMF models are modified directly without prior copying or conversion. HENSHIN is able to handle static and dynamic EMF models, i.e., those with underlying generated model code and those without. The transformation concepts base on the well-founded theory of algebraic graph transformation with pattern-based rules as main artifacts, extended by nestable application conditions and attribute calculation. Moreover, nestable transformation units with well-defined operational semantics paired with parameter passing allow to define control and object flows. In the HENSHIN tool environment, transformations can be specified using several (graphical) editors.

In the following, a representative selection of the complete solution of the Transformation Tool Contest (TTC) 2011 case study Model Transformations for Program Understanding: A Reengineering Challenge is described. The goal of this case study is to translate JaMoPP-based Java models into corresponding simple state machine models. This translation is implemented using HENSHIN.

2 EMF Model Transformation with HENSHIN

HENSHIN’s transformation meta-model is an EMF model itself. As one of its core concepts, transformation rules consist of a left-hand side (LHS), describing the pattern to be matched, and a right-hand side (RHS), describing the resulting pattern. Node mappings between the LHS and the RHS declare identity, i.e., such nodes are preserved. Rules may also have positive and negative application conditions (PACs and NACs, respectively) specifying additional constraints over the match. Moreover, application conditions can be combined using standard Boolean operators (NOT, AND, OR), which facilitates an arbitrary nesting of conditions. Attribute calculations are evaluated at runtime by Java’s built-in JavaScript engine which may also call Java methods.

Predefined nestable transformation units allow to control the order of rule application. Note that rules are considered to be atomic units corresponding to their single application. Independent units provide...
a non-deterministic choice, priority units allow to specify prioritized unit applications, counted applications are provided by the counted unit with a count value of -1 meaning “as often as possible”. Sequential units apply units sequentially while performing a rollback if an application fails, and conditional units allow to specify an if condition with corresponding then and else parts. So-called amalgamation units represent a forall-operator for pattern replacement at which a kernel rule is matched once and arbitrary multi rules are each matched as often as possible in the context of the kernel rule’s match.

Typeless parameters and parameter mappings from one unit to others specify object flows and enable to pre-define (partial) matches.

Currently, three different editors provide three different views on HENSHIN transformation models. The tree-based editor provides a linear and low-level view on the internal model structure, while two other editors offer a more sophisticated graph-like visualization: One visual editor, called complex-rule editor, shows LHS, RHS and application conditions in separate views whereas the integrated-rule editor depicts rules in an integrated manner using a single view and utilizing stereotypes to denote creation, deletion and preservation. Although the complex-rule editor is particularly suitable for complex transformation systems with arbitrary control and object flows, in the following rules and units are illustrated using the tree-based and especially the integrated-rule editor due to its concise representation.

Rules and units may be applied on arbitrary EMF models by a dedicated wizard or by Java code. HENSHIN comes with an independent transformation engine which can be freely integrated in any Java project relying on EMF models. A convenient API provides classes such as RuleApplication and UnitApplication for the selection and application of rules and units, respectively.

For more information we refer to the solution of the Hello World instructive case [4].

3 The Solution

In the following, a subset of the complete solution of the reengineering challenge [3] is presented while a full listing of rules and transformation units is given in Appendix A. Java source code triggers the transformation which can be found in Appendix B. Since HENSHIN currently does not support list semantics but set semantics only, we exploit a self-contained helper structure called trace model to simulate iteration by marking already processed elements. This model is part of HENSHIN and consists of a class Trace with two generic outgoing references source and target.

Start. The JaMoPP to state machine model transformation is performed by executing a single sequential unit, Start, shown in Fig. 1 by means of the tree-based editor. Start contains[1] the rule init performing prerequisites and three counted units StatesLoop(count=-1), TransitionsLoop(count=-1) and ActionsLoop(count=-1) dealing with the creation of :StateMachine and :Class objects. The core task and the extension task 1 are realized by the first two counted units, and extension task 2 is implemented by the latter. The parameters sm and class are initially empty and represent the :StateMachine root object to be created and the :Class instance named “State”, respectively. Particularly, sm is used to persist the state machine model after the transformation has finished. Note that parameter mappings are not visualized throughout this paper in favor of conciseness and readability. The

---

1In fact, all rules and units are structurally contained in a :TransformationSystem root object but they may be referred to by other units allowing reuse. Referencing is denoted by small arrows in the bottom-right of the icons of rules and units.
reader may primarily assume equally named parameters being mapped top-down, i.e., from containing units to contained units/rules.

Figure 2 shows the rule init in the integrated-rule editor. A rule is presented as rounded rectangle with its name at the top followed by owning parameters and its graph structure contained. Stereotypes denote nodes and edges to be created, deleted, preserved or forbidden. Parameters may appear in front of node typings or as attribute values in order to represent an object or a value. Optional keywords in square brackets indicate inbound and outbound parameters. No identifier means in and out. Unset parameters are set during the matching while predefined parameters limit valid matches. The rule init creates a :StateMachine object and matches a :Class named “State”. Both objects are then stored in the outbound parameters sm and class which finally pass the values to related parameters of the enclosing unit Start due to parameter mapping contained in Start.

States. The next step is to create all :State objects which is performed by the counted unit StatesLoop in a recursive manner. In the left of Fig. 3 the related control structure is given. At its first invocation, StatesLoop receives the value of class of Start pointing to class “State”. The priority unit CreateState-AndChildren tries to apply createState (see top right of Fig. 3) as often as possible. The rule createState matches only if the :Class given by parameter class is not abstract and no equally named :State is available which is equivalent to “already translated”. If both constraints hold, a new :State object is created and added to the existing :StateMachine object. Otherwise, conditional unit ProcessChildren is executed to retrieve a child class of class by applying the rule checkClassHasChild (see bottom right of Fig. 3) in its if condition. Consequently, the rule checkClassHasChild takes parameter class into account as well and matches a child class that has not been marked yet by a :Trace object. If such child class exists, it is marked and returned via parameter child which is mapped to ProcessChildren’s child parameter. Furthermore, the recursion is performed by calling unit StatesLoop whose class parameter is set to the value of the current child. If neither createState nor checkClassHasChild could be applied, the
exit condition is fulfilled. Note that each call of a transformation unit spawns its own set of parameters with related values, which is why recursion is possible.

Transitions and Triggers. After all state classes have been translated to :State objects, the creation of transitions between them is accomplished by applying the content of the counted unit TransitionsLoop as often as possible. Figure 4 shows the main control flow which is an iteration over all translated classes and for each class an iteration over all associated :StateMachine objects. Note that parameters are left out in order to focus on the control flow.

The actual creation of transitions is performed by the counted unit DescendLoop (cf. Fig. 4 & 5) which is similarly structured to StatesLoop (cf. Fig. 3) since both work recursively. In the loop, the priority unit CreateOrDescend tries to apply the rule createTransition or alternatively tries to execute the conditional unit Descend. createTransition is depicted in the right of Fig. 5 exposing a number of parameters used. Its parameter baseClass identifies the current class in the iteration that has been passed down since rule nextClass. parent identifies the current element in the recursion process and is here required to be of type ExpressionStatement. The parameters srcName and trgName are used within the rule only and ensure that the names of target and source states correspond to the base class and the class being part of the expression. Finally, parameter trigger contains the trigger value collected beforehand by rule nextClassMethod or in a previous recursion step by descendTryCatch or descendSwitch and is consequently passed along the control flow. If the rule can be applied, a :Transition object is created with a default action attribute value and a trigger attribute value evaluated depending on the value of parameter trigger. In addition, a :Trace object is created and associated, on the one hand, to mark the :ExpressionStatement as being visited and, on the other hand, to assist the rule updateActions which is part of extension task 2 (see below). If createTransition cannot be applied, the unit TryDescending in the if condition of Descend performs a single descending step along the structure of a :StateMachine object. On success, a recursive execution of unit Descend-
Loop is performed in the then part, otherwise, the actual execution of DescendLoop finishes. Since HENSHIN does not support typeless references or path expressions presently, each descending case needs to be modeled separately, e.g., descendTryCatch, descendSwitch, etc. Nevertheless, parameters are typeless which allows parameter child of the unit TryDescending to store any object found in a descending step.

Actions. Since all :Trigger objects are already equipped with a default action attribute value (see above), the counted unit ActionsLoop and its single contained rule updateAction only need to update specific transitions. For this purpose, the rule updateActions matches a structure corresponding to a call to send() and also a related :ExpressionStatement object which has been marked by a :Trace object in the rule createTransition. On rule application, the action attribute value is updated and the :Trace object is removed to prevent double matchings.

4 Conclusion

In this paper, the HENSHIN solution to the TTC 2011 Reengineering case [3] is presented. It covers all tasks including the extension tasks 1 and 2. The implementation is made available under SHARE [5].

The solution is particularly characterized by a visual transformation language, pattern-based rules and control and object flows. Furthermore, cyclic (recursive) control flows have been exploited to efficiently walk along tree-like graph structures. Note that this solution is a heavily optimized version of the one presented at the workshop where no cyclic control flow had been used and a significant higher number of rules and transformation units were required.

The HENSHIN tool environment offers a number of different editors, each one suited better for a specific task. However, switching between different editors is not optimal. Therefore, we plan to provide a single feature-complete editor in the next major release of HENSHIN. For this purpose, we intend to provide a DSL for a convenient editing. Furthermore, since EMF primarily employs lists instead of sets, we plan to extend Henshin by related control structures in order to make the costly use of additional trace objects obsolete. Nevertheless, with a time consumption of < 1sec, < 1sec and ~ 5sec (Core2Duo 2Ghz) for a transformation of the small, medium and big example models, respectively, the solution performs sufficiently fast in our opinion.

References

[1] T. Arendt, E. Biermann, S. Jurack, C. Krause & G. Taentzer (2010): Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformation. In: Proc. of 13th Int. Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), LNCS, Springer, pp. 121–135, doi:10.1007/978-3-642-16145-2_9.

[2] Henshin (2010): [http://www.eclipse.org/modeling/emft/henshin](http://www.eclipse.org/modeling/emft/henshin).

[3] Tassilo Horn (2011): Program Understanding: A Reengineering Case for the Transformation Tool Contest. In Van Gorp et al. [7].

[4] Stefan Jurack & Johannes Tietje (2011): Saying Hello World with Henshin - A Solution to the TTC 2011 Instructive Case. In Van Gorp et al. [7].

[5] Stefan Jurack & Johannes Tietje (2011): SHARE demo related to the paper Solving the TTC 2011 Reengineering Case with Henshin. [http://is.tm.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_Henshin.vdi](http://is.tm.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_Henshin.vdi).

[6] Dave Steinberg, Frank Budinsky, Marcelo Paternostro & Ed Merks (2009): EMF Eclipse Modeling Framework (Second Edition). Addison-Wesley.
A All Solutions

In the following, the complete solution is presented with rules visualized by means of the integrated-rule editor, and control and object flows shown by means of the tree-based editor.

![Figure 6: Sequential unit Start being the entry point of the transformation. Parameters and parameter mappings are also shown at which external source or target parameters of mappings are denoted by their owning transformation unit’s name and the parameter name, e.g., init.sm. Note that parameters of rules are not shown in this and the following tree-based figures although the tree-based editor provides them to the user. However, they are shown in the integrate-rule presentation.](image)

![Figure 7: The first rule applied at all: init. It contains the parameters sm and class which occur in the RHS only and therefore may only be used as output.](image)
Figure 8: Sequential unit *StatesLoop* ensures the translation of children of class \(\text{Class(name='State')}\) into \(\text{State}\).

Figure 9: Rule *createState* creates a \(\text{State}\) object related to a \(\text{Class}\) being a child of \(\text{Class(name='State')}\). Negative application conditions denoted by stereotype \<<\text{forbid}\>>\) ensure that only non-abstract classes are translated and that no class is translated twice.
Figure 10: Rule `checkClassHasChild` matches a child of the :Class specified by parameter `class`. The child must not be visited/matched twice which is ensured by a :Trace object.

```
checkClassHasChild(class,child)
```

- `checkClassHasChild` classifies the relationship between `class` and `child`.
- `child[out]:Class` is the child of `class`.
- `ClassifierReference` is preserved.
- `source` is forbidded to be visited.
- `classifierReferences` and `source` are created.
- `extends` and `target` are preserved.

Figure 11: Sequential unit `TransitionsLoop` deals with the creation of transitions between :State objects related to specific method calls between classes. Note that in unit `ProcessClassMethod` parameter mappings are arranged in a dedicated group “Parameter Mappings” which is the default visualization for more than four parameter mappings in a unit. Note furthermore that unit `DecendLoop` is fold and shown in Fig. 14 below.

```
Sequential Unit CreateTransitions
- Parameter baseClass
- Parameter Mapping baseClass -> ClassMethodsLoop.baseClass
- Parameter Mapping nextClass.c -> baseClass

Rule `nextClass`
- Counted Unit ClassMethodsLoop
  - Parameter baseClass
    - Parameter Mapping baseClass -> ProcessClassMethod.baseClass
  - Sequential Unit ProcessClassMethod
    - Parameter baseClass
    - Parameter classMethod
    - Parameter trigger
    - Parameter Mappings
      - Parameter Mapping baseClass -> nextClassMethod.baseClass
      - Parameter Mapping nextClassMethod.cm -> classMethod
      - Parameter Mapping classMethod -> DescendLoop.current
      - Parameter Mapping baseClass -> DescendLoop.baseClass
      - Parameter Mapping trigger -> DescendLoop.trigger
      - Parameter Mapping nextClassMethod.cmName -> trigger

Rule `nextClassMethod`
- Counted Unit DescendLoop
```
Figure 12: Rule `nextClass` matches a `:State` and its corresponding `:Class` object. This ensures that only such class is found which is a non-abstract child of `:Class(name='State')` since only they were translated to `:States`. The child found is provided to the environment by parameter `c`. Again, a `:Trace` object which is created and also forbidden to exist ensures that a state (and also its related class) is matched only once.

Figure 13: Rule `nextClassMethod` matches a `:ClassMethod` object associated with a given `:Class` which is predefined by parameter `baseClass`. The `:ClassMethod` itself and its name are provided to the environment by the parameters `cm` and `cmName`. Note that `cmName` is used as part of extension task 1 and retrieves the name of the method in order to set the `trigger` attribute value of the transition to be created later (see parameter mapping `nextClassMethod.cmName → trigger` in Fig. 11).
Figure 14: Sequential unit `DescendLoop` is part of unit `TransitionsLoop` (see Fig. [11]) and deals with the creation of transitions. The control flow is defined cyclic (recursive), i.e., in unit `Descend` a call to the enclosing unit `DescendLoop` is performed shown at the very bottom of this figure. While the whole algorithm defined by this unit is pretty simple, its representation appears confusing due to the number of parameters and parameter mappings. This is a clear shortcoming of HENSHIN currently and will be fixed in the near future. Note that a unit, `TryDescending`, is still fold and presented below in Fig. [16]
createTransition(baseClass, parent, trigger, srcName, trgName)

Figure 15: Rule `createTransition` creates the :Transition object including its trigger and action attribute values. The transition is created in relation to an :ExpressionStatement which has been handed over by parameter `parent`. Parameters `srcName` and `trgName` are not predefined but are set during matching and ensure the correct matching of corresponding classes and states. The trigger attribute is evaluated on the basis of the value of parameter `trigger`. In contrast, the action attribute is set to a default value. Again, :Trace objects ensure that each corresponding structure is matched only once. Furthermore, it links the :ExpressionStatement with the newly created :Transition in order to ease the extension task 2, i.e., the setting of the correct action attribute value performed in rule `updateAction` (see Fig. 19).
Figure 16: Priority unit *TryDescending* is part of unit *DecendLoop* (see Fig. 14) and performs the descending of the :MethodClass structure. Since HENSHIN does not support neither path expressions nor untyped nodes and untyped edges, each case has to be handled by a single rule separately. The priority unit makes sure, that the first applicable rule is applied. Parameter mappings running to current then return a child to be used later for the recursive call. Two rules, *descendTryCatch* and *descendSwitch*, return and thus update the current trigger value of the enclosing unit(s). While the parameter mappings look confusing at first sight, having a closer look reveals a recurring mechanism, i.e., for each rule the parameter current is set and the returned child adopted.
Figure 17: Rules descendSLC, descendSC, descendCondition, and descendTryFinally being part of the top-down traversal of the tree-like structure with :ClassMethod (being a subtype of StatementListContainer) as top-most element.
Figure 18: Rules \texttt{descendTryCatch} and \texttt{descendSwitch} performing a top-down traversal analog to the rules in Fig. [17]. In addition, the trigger value is fetched to be used in the subsequent creation of a transition (see Fig. [15]).
Figure 19: Counted unit ActionsLoop (left) and rule updateAction (right). As often as possible, the rule updates the action attribute value of any Transition being associated by a Trace object which points to an ExpressionStatement and which in turn is contained by a StatementListContainer. The Trace object previously created in rule createTransition (see Fig. [15]) is deleted during the application of this rule. This ensures that no transition is updated twice.
B Java Code of the Transformation Application

This section shows the code that triggers the transformation. In addition to that, the following listing contains code to measure the time spend and code to check the correctness of the solution using EMF Compare (http://www.eclipse.org/emf/compare/).

Listing 1: Starter for the transformation.

```java
package de.jtietje.fh.ma.ttc2011;

import java.io.File;
import java.io.IOException;
import java.util.Collections;
import java.util.List;

import mapping.impl.MappingPackageImpl;
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.compare.diff.metamodel.DiffElement;
import org.eclipse.emf.compare.diff.metamodel.DiffModel;
import org.eclipse.emf.compare.diff.service.DiffService;
import org.eclipse.emf.compare.match.metamodel.MatchModel;
import org.eclipse.emf.ecore.EObject;
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.ecoreFactoryImpl;
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.TransformationSystem;
import org.eclipse.emf.henshin.model.TransformationUnit;
import org.eclipse.emf.henshin.model.impl.HenshinPackageImpl;
import org.eclipse.emf.henshin.model.resource.HenshinResourceFactory;
import org.eclipse.emf.henshin.trace.impl.TracePackageImpl;
import org.emftext.language.java.containers.CompilationUnit;
import org.emftext.language.java.containers.ContainersFactory;
import org.emftext.language.java.containers.Package;
import org.emftext.language.java.impl.JavaPackageImpl;

import statemachine.StateMachine;
import statemachine.impl.StatemachinePackageImpl;

/**
 * M2M transformation with a JaMoPP Java model as source
 * model and a state machine model as target. This is an
 * implementation in terms of case study 1 of TTC2011
 * (http://planeteresearch20.org/ttc2011/index.php?option=
 * com_content&view=article&id=118&Itemid=160).
 */
```
public final class JaMoPP2StateMachine {

    /**
     * The resource set for loading XMI files.
     */
    private static final ResourceSet RESOURCESET = new ResourceSetImpl();

    /**
     * time measurement purpose
     */
    private static long stopwatchValue = 0;

    /**
     * Program entry point.
     * @param args the command line arguments
     */
    public static void main(final String[] args) {
        if (args.length != 1) {
            System.err
                .println("Usage: TCPStateExtract {1_small-model, 2_medium-model, 3_big-model}");
            System.exit(-1);
        } // if

        initializeFactories();
        loadAndTransform(args[0]);
    } // main

    /**
     * Loads and transforms the JaMoPP model with the given path.
     * @param model model to transform
     */
    private static void loadAndTransform(final String model) {
        String modelFile = "model/source-models/" + model + ".xmi";
        String henshinFile = "henshin/statemachine2.henshin";
        String stateMachineXMIFile = "model/generated/" + model + ".statemachine";

        Package rootPackage = loadJaMoPPSourceModel(modelFile);
        StateMachine stateMachine = performTransformation(
            henshinFile, rootPackage);
        saveModel(stateMachine, stateMachineXMIFile);
```java
private static Package loadJaMoPPSourceModel(final String path) {

    JaMoPP2Statemachine.stopwatch("Start loading the model...");
    Resource xmiResource = loadModel(path);
    EList<EObject> resourceContents = xmiResource.getContent();

    Package rootPackage = ContainersFactory.eINSTANCE.createPackage();
    List<CompilationUnit> compilationUnits = rootPackage.getCompilationUnits();

    for (EObject compilationUnit : resourceContents) {
        if (compilationUnit instanceof CompilationUnit) {
            compilationUnits.add((CompilationUnit) compilationUnit);
        }
    }

    JaMoPP2Statemachine.stopwatch("Time to load the model: ");

    return rootPackage;
}
```

```java
/**
 * Compare result with reference model
 */
String referenceFile = "model/statemachine/reference.statemachine";
compareEMF(referenceFile, stateMachineXMIFile);
} // loadAndTransform

/**
 * Loads a given JaMoPP source code model. All contained
 * {@link CompilationUnit}s are attached to a newly
 * created {@link Package} as single root element. This is
 * for convenience only.
 *
 * @param path
to the source code model
 * @return a newly created {@link Package} with
 * compilation units contained.
 */
private static Package loadJaMoPPSourceModel(final String path) {

    JaMoPP2Statemachine.stopwatch("Start loading the model...");
    Resource xmiResource = loadModel(path);
    EList<EObject> resourceContents = xmiResource.getContent();

    Package rootPackage = ContainersFactory.eINSTANCE.createPackage();
    List<CompilationUnit> compilationUnits = rootPackage.getCompilationUnits();

    for (EObject compilationUnit : resourceContents) {
        if (compilationUnit instanceof CompilationUnit) {
            compilationUnits.add((CompilationUnit) compilationUnit);
        }
    }

    JaMoPP2Statemachine.stopwatch("Time to load the model: ");

    return rootPackage;
}
```

```java
/**
 * Loads the henshin transformation model .
 *
 * @param path
 * the path of the Henshin file
 *
 * @param rootPackage
 * a {@link Package} containing a list of
 * compilation units
 *
 * @return a state machine object representing the result
 * of the transformations
 */
```
private static StateMachine performTransformation(
    final String path, final Package rootPackage) {

    JaMoPP2Statemachine
        .stopwatch("Start preparing the transformation ...")
    Resource xmiResource = loadModel(path);
    TransformationSystem transformationSystem = (TransformationSystem)
        xmiResource
            .getContents().get(0);
    TransformationUnit transformationUnit = transformationSystem
        .findUnitByName("Start");
    // internal representation of the EMF model
    EmfGraph emfGraph = new EmfGraph();
    emfGraph.addRoot(rootPackage);  
    EmfEngine emfEngine = new EmfEngine(emfGraph);  
    UnitApplication unitApplication = new UnitApplication(  
        emfEngine, transformationUnit);
    JaMoPP2Statemachine.stopwatch("Time for preparations: ");

    JaMoPP2Statemachine
        .stopwatch("Start performing the transformation ...");
    if (unitApplication.execute()) {
        System.out.println("Successful.");
    } else {
        System.out.println("Not successful.");
    } // if else
    JaMoPP2Statemachine.stopwatch("Time for transformation: ");

    return (StateMachine) unitApplication
        .getParameterValue("sm");
} // applyHenshinRules

AMESPACE
    * Loads an EMF model file and returns it as a Resource.
    * @param modelPath
    *   the path to the model file
    * @return the Resource representing the model file
    */
private static Resource loadModel(final String modelPath) {
    URI modelUri = URI.createFileURI(new File(modelPath)
        .getAbsolutePath());
    return RESOURCESET.getResource(modelUri, true);
} // loadModel

AMESPACE
    * Serializes a given EMF model into an XMI file of the
    * given path.
    * @param model
    *     the model to serialize
    */
/* @param path
* the path of the resulting file */

private static void saveModel(final EObject eobject,
                          final String path) {
    URI ecoreUri = URI.createFileURI(new File(path)
          .getAbsolutePath());

    Resource ecoreResource = RESOURCESET
          .createResource(ecoreUri);
    ecoreResource.getContents().add(eobject);

    try {
        ecoreResource.save(null);
    } catch (IOException e) {
        e.printStackTrace();
    }
} // saveStateMachine

/**
* Initializes related factories and packages.
*/

private static void initializeFactories() {
    Resource.Factory.Registry.INSTANCE
          .getExtensionToFactoryMap().put("ecore",
                new EcoreResourceFactoryImpl());
    Resource.Factory.Registry.INSTANCE
          .getExtensionToFactoryMap().put("xmi",
                new XMIResourceFactoryImpl());
    Resource.Factory.Registry.INSTANCE
          .getExtensionToFactoryMap().put("henshin",
                new HenshinResourceFactory());
    Resource.Factory.Registry.INSTANCE
          .getExtensionToFactoryMap().put("statemachine",
                new XMIResourceFactoryImpl());

    JavaPackageImpl.init();
    HenshinPackageImpl.init();
    StatemachinePackageImpl.init();
    MappingPackageImpl.init();
    TracePackageImpl.init();
} // initializeFactories

/**
* This method is for time measurement only and shall be
* ignored to understand the actual transformation
* process.
*/

private static final void stopwatch(final String message) {
    if (stopwatchValue == 0) {
        stopwatchValue = System.currentTimeMillis();
    } else {  // try catch
System.out.println(message);
stopwatchValue = System.nanoTime();
}
else {
    long temp = System.nanoTime();
    System.out.format("%.4f sec \n", message,
        ((float)(temp - stopwatchValue)) / 1000000000);
    stopwatchValue = 0;
} // if else
} // stopwatch

/**
 * Compares two given EMF files via EMF Compare. This
 * method is only to check the result of the
 * transformation, thus it is not part of the
 * transformation process itself.
 * @param referenceFile
 * the source model file
 * @param compareFile
 * the target model file
 */
private static void compareEMF(final String referenceFile,
        final String compareFile) {
    Resource leftResource = RESOURCESET.getResource(
            URI.createFileURI(compareFile), true);
    Resource rightResource = RESOURCESET.getResource(
            URI.createFileURI(referenceFile), true);
    MatchModel match = null;
    try {
        match = (new StatemachineMatcher()).resourceMatch(
                leftResource, rightResource,
                Collections.<String, Object>emptyMap());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } // try catch
    DiffModel diff = DiffService.doDiff(match, false);
    System.out.println("Printing the differences...");  
    for (DiffElement diffElement : diff.getDifferences()) {
        System.out.println(diffElement.toString());
    } // for
} // compareEMF

} // class
Listing 2: Helper class for comparing the correctness of the result with EMF Compare.

```java
package de.jti.etje.fh.ma.ttc2011;

import java.util.List;

import org.eclipse.emf.compare.FactoryException;
import org.eclipse.emf.compare.match.engine.GenericMatchEngine;
import org.eclipse.emf.ecore.EObject;

import statemachine.State;
import statemachine.Transition;

/
* A custom matcher for statemachine model elements as
* helper class for EMF Compare. This is used to compare the
* result of the JaMoPP2StateMachine transformation with an
* exemplar given by the TTC host.
* /

public class StatemachineMatcher extends GenericMatchEngine {

    @Override
    protected final EObject findMostSimilar(final EObject eObj, final List<EObject> list) throws FactoryException {
        if (eObj instanceof Transition) {
            Transition source = (Transition) eObj;

            for (EObject eObject : list) {
                if (eObject instanceof Transition) {
                    Transition potentialMostSimilar = (Transition) eObject;
                    boolean isMostSimilar = isSimilar(source.getSource(),
                                                      potentialMostSimilar.getSource())
                        && isSimilar(source.getDestination(),
                                    potentialMostSimilar.getDestination());

                    if (isMostSimilar) {
                        return potentialMostSimilar;
                    }
                }
            }
        }

        return super.findMostSimilar(eObj, list);
    }

    @Override
    protected final boolean isSimilar(final EObject obj1, final EObject obj2) throws FactoryException {
        if (obj1 instanceof State || obj2 instanceof State) {
            State firstState = (State) obj1;
        }
    }
```
State secondState = (State) obj2;
return firstState.getName().equals(secondState.getName());
}
else {
if (obj1 instanceof Transition
    || obj2 instanceof Transition) {
    Transition firstTransition = (Transition) obj1;
    Transition secondTransition = (Transition) obj2;

    boolean preCheck = firstTransition.getAction().equals(
        secondTransition.getAction())
        && firstTransition.getTrigger().equals(
        secondTransition.getTrigger());
    if (preCheck) {
        return isSimilar(firstTransition.getDst(),
            secondTransition.getDst())
            && isSimilar(firstTransition.getSrc(),
            secondTransition.getSrc());
    } else {
        return false;
    }
}
return super.isSimilar(obj1, obj2);