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) [6] 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 [1, 2] 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* [3] 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

Van Gorp, Mazanek and Rose (Eds.):
Fifth Transformation Tool Contest (TTC 2011)
EPTCS 74, 2011, pp. 181–203, doi:10.4204/EPTCS.74.17

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`.

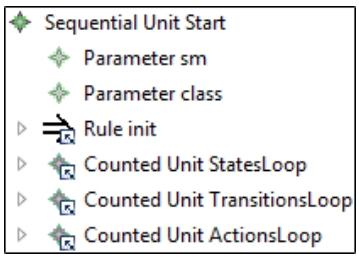

Figure 1: Outline.

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 `:State` and `:Transition` 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

[1] In 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.

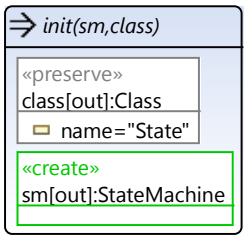

Figure 2: Rule *init*.

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 *CreateStateAndChildren* 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

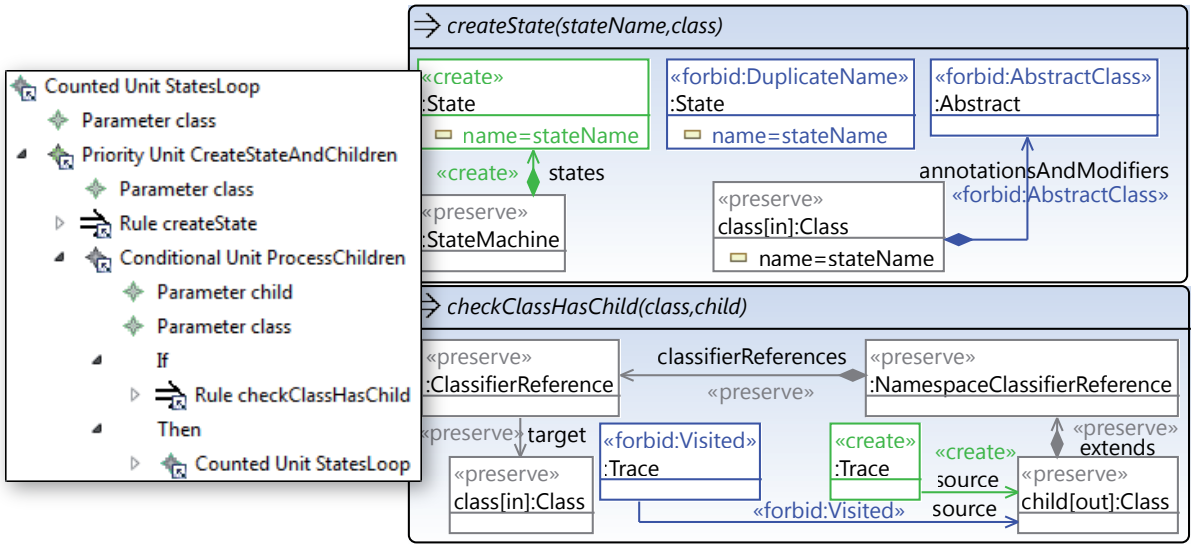

Figure 3: Control flow (left) and the key rule *createState* (right) for the translation of classes to states.

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.

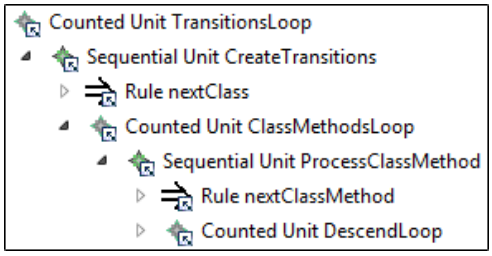

Figure 4: Unit *TransitionsLoop*.

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 `:ClassMethod` 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 `:ClassMethod` object. On success, a recursive execution of unit *Descend-*

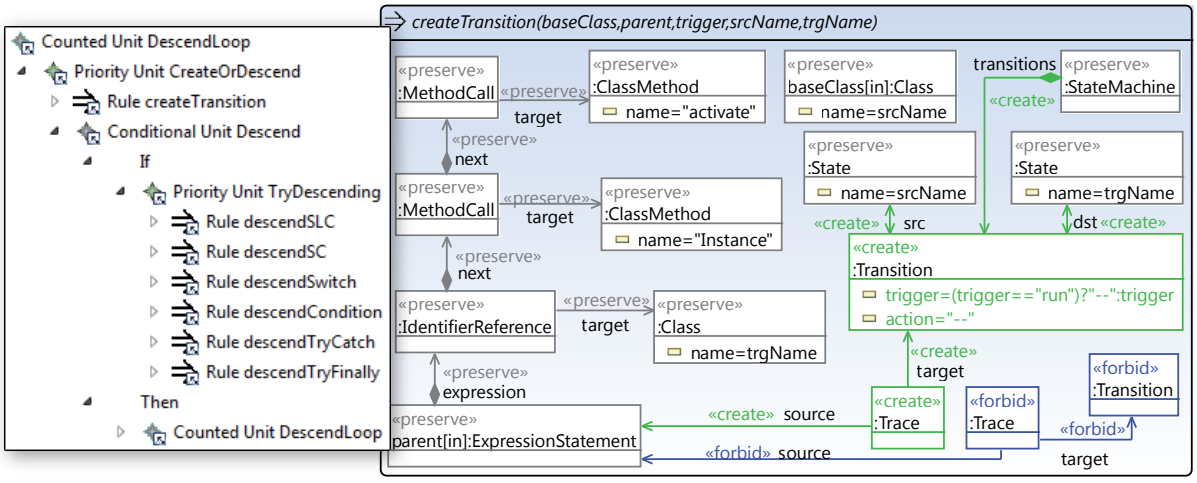

Figure 5: Control flow (left) and key rule *createTransition* (right) for the creation of a transition.

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 $\sim 5sec$ (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`.

[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`.

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

[7] Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors (2011): *TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011, Post-Proceedings*. EPTCS.

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.

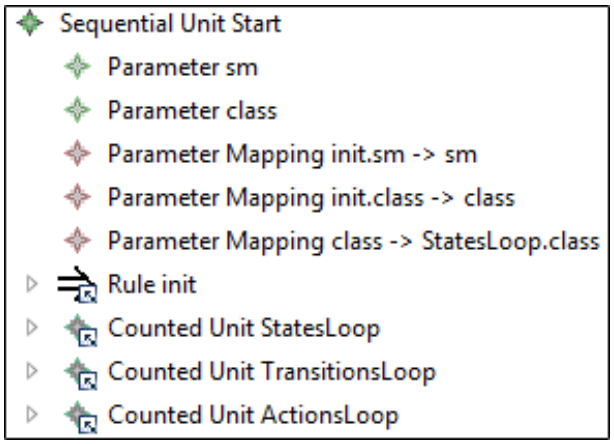

Figure 6: Sequential unit *Start* being the entry point of the transformation. Parameters and parameter mappings are also shown at which external source or target parameters of mappings are denoted by their owning transformation unit's name and the parameter name, e.g., *init.sm*. Note that parameters 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.

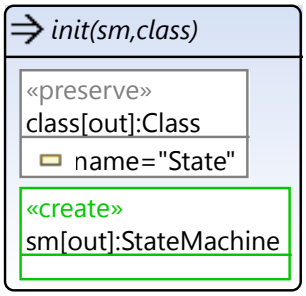

Figure 7: The first rule applied at all: *init*. It contains the parameters *sm* and *class* which occur in the RHS only and therefore may only be used as output.

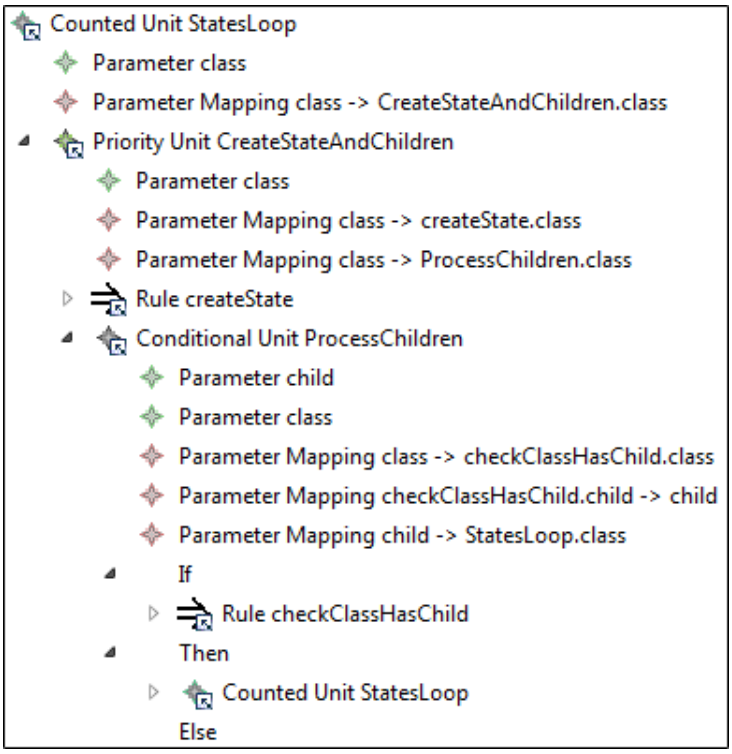

Figure 8: Sequential unit *StatesLoop* ensures the translation of children of class `:Class(name=''State'')` into `:States`.

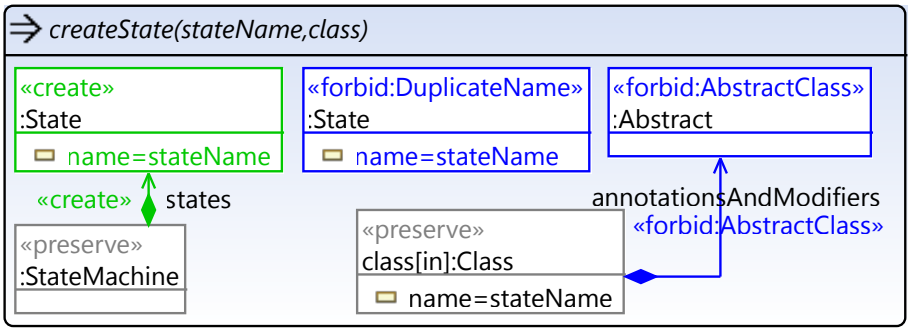

Figure 9: Rule *createState* creates a `:State` object related to a `:Class` being a child of `:Class(name=''State'')`. Negative application conditions denoted by stereotype `<<forbid>>` ensure that only non-abstract classes are translated and that no class is translated twice.

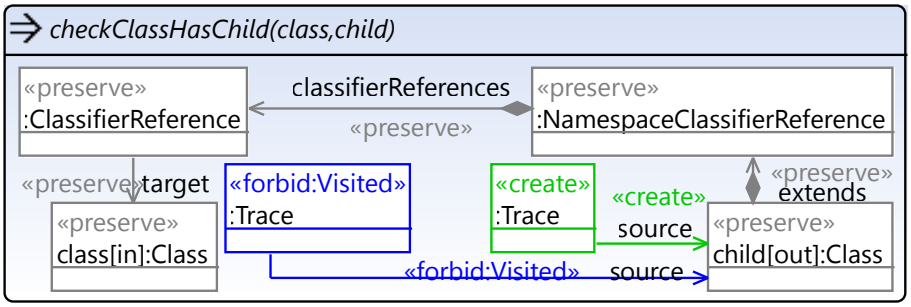

Figure 10: Rule *checkClassHasChild* matches a child of the `:Class` specified by parameter *class*. The child must not be visited/matched twice which is ensured by a `:Trace` object.

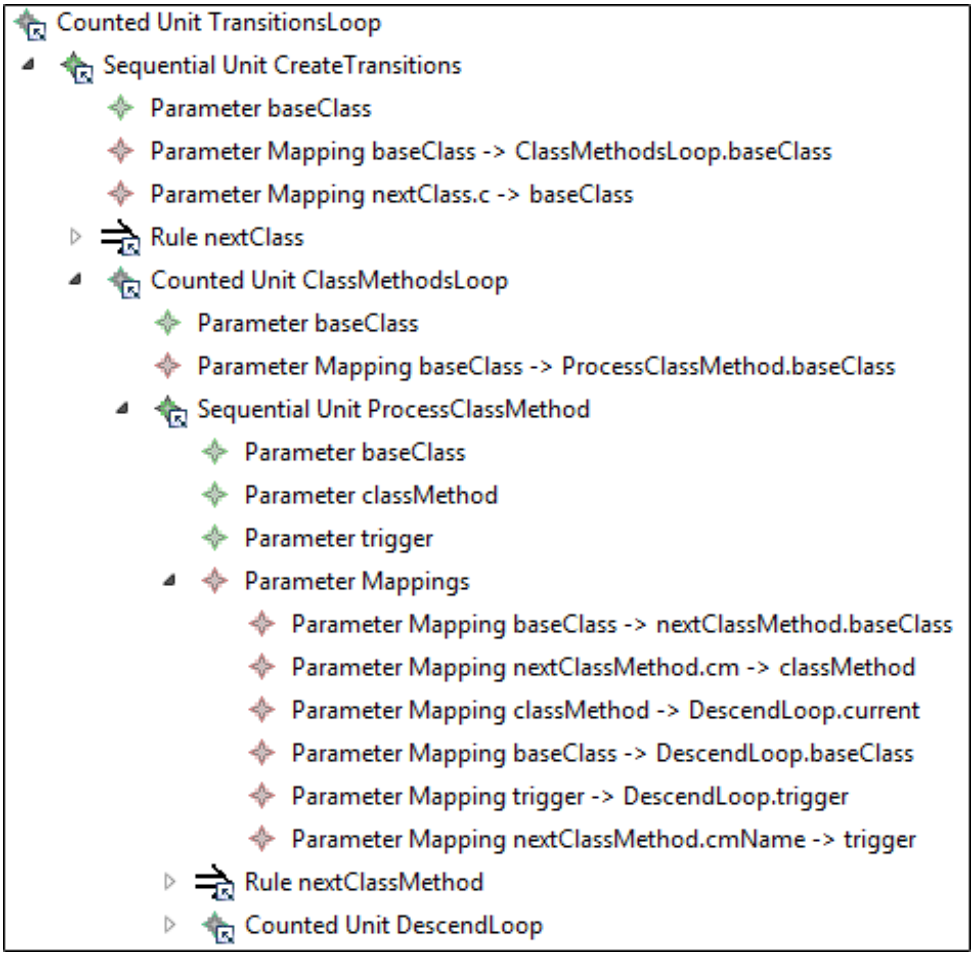

Figure 11: Sequential unit *TransitionsLoop* deals with the creation of transitions between `:State` objects related to specific method calls between classes. Note that in unit *ProcessClassMethod* parameter mappings are arranged in a dedicated group "Parameter Mappings" which is the default visualization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

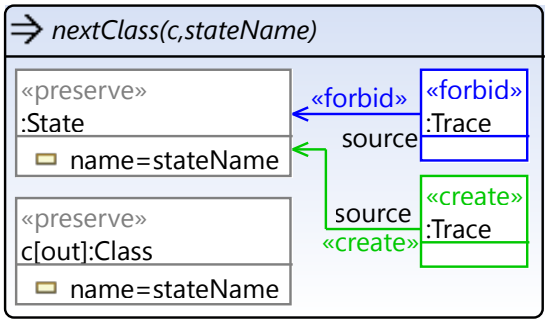

Figure 12: Rule *nextClass* matches a `:State` and its corresponding `:Class` object. This ensures that only such class is found which is a non-abstract child of `:Class(name=''State'')` since only they were translated to `:States`. The child found is provided to the environment by parameter *c*. Again,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

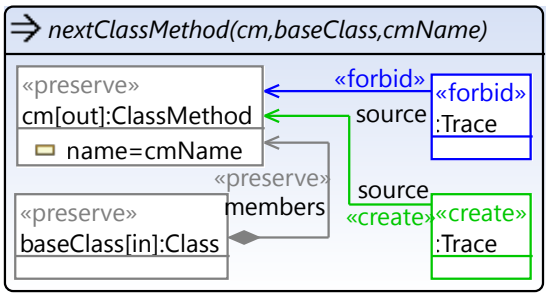

Figure 13: Rule *nextClassMethod* matches a *:ClassMethod* object associated with a given `:Class` which is predefined by parameter *baseClass*. The `:ClassMethod` itself and its name are provided to the environment by the parameters *cm* and *cmName*. Note that *cmName* is used as part of extension 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).

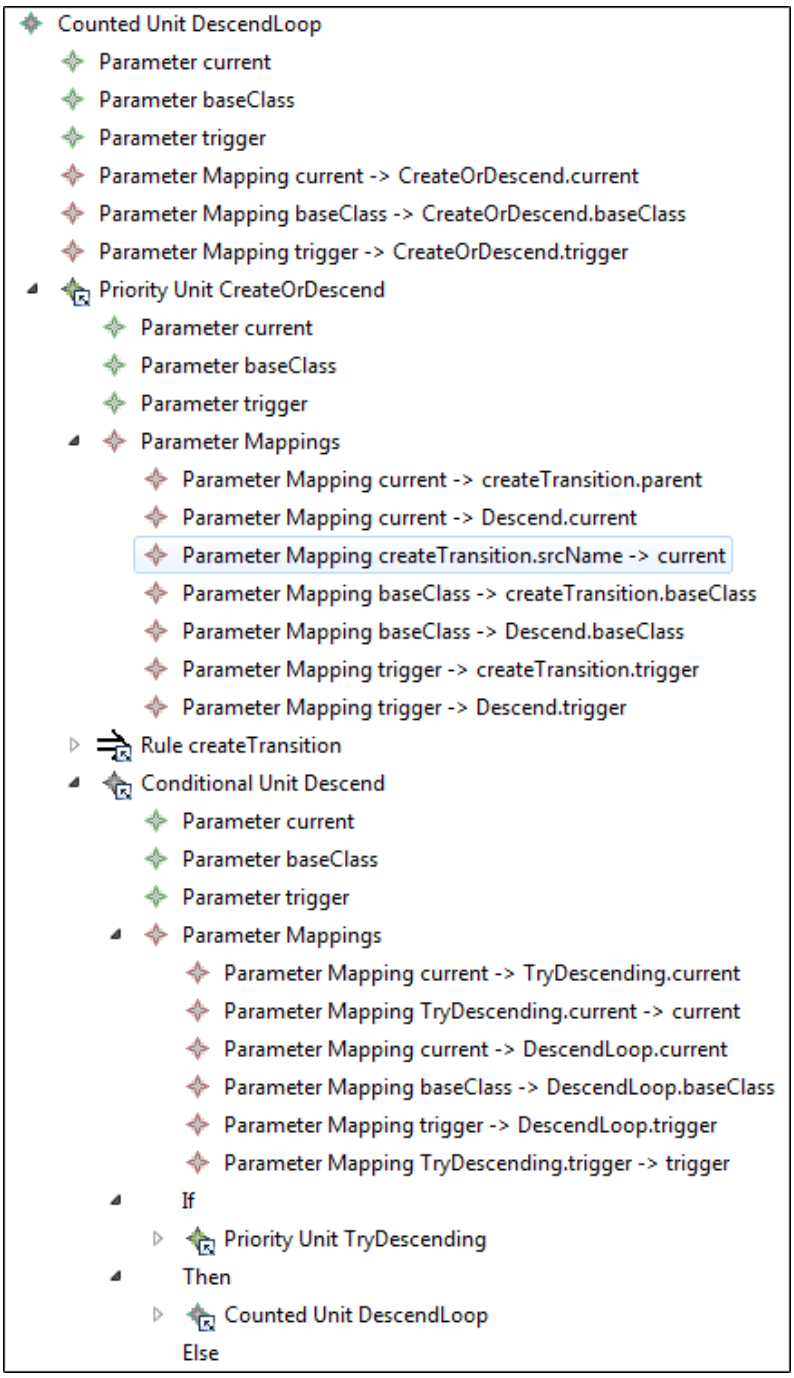

Figure 14: Sequential unit *DescendLoop* is part of unit *TransitionsLoop* (see Fig. 11) and deals with the creation of transitions. The control flow is defined cyclic (recursive), i.e., in unit *Descend* a call to the enclosing unit *DecendLoop* is performed shown at the very bottom of this figure. 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.

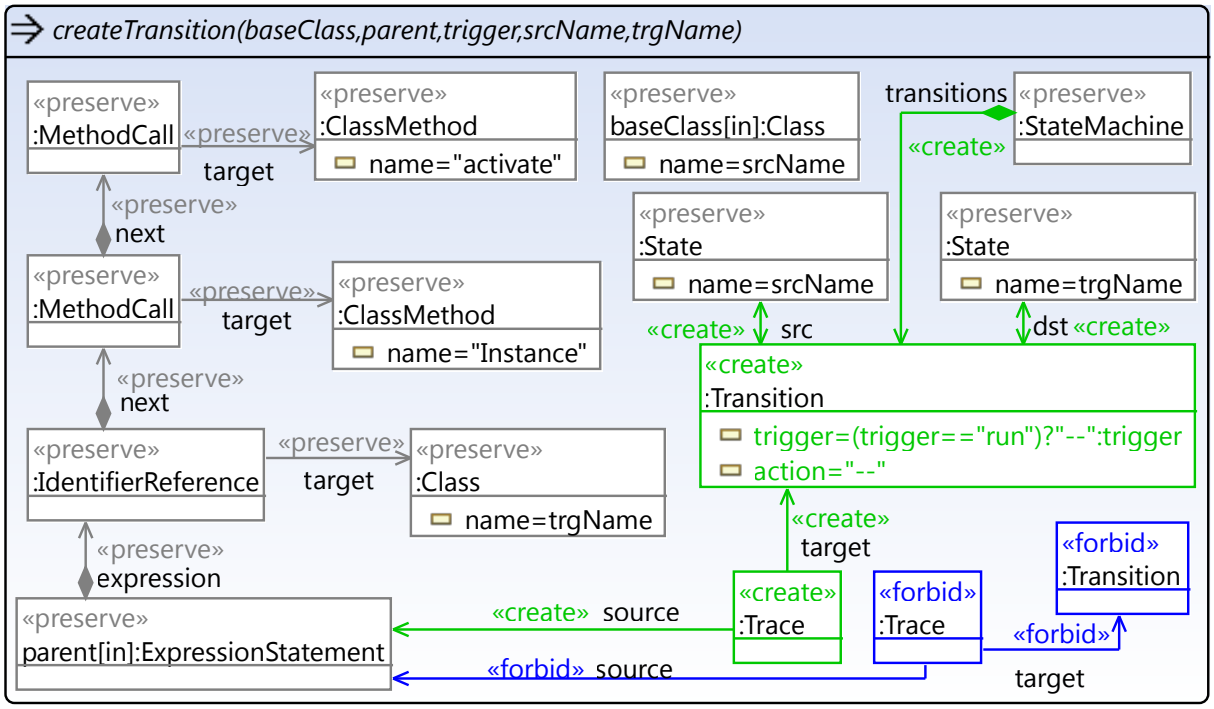

Figure 15: Rule *createTransition* creates the `:Transition` object including its `trigger` and `action` attribute values. The transition is created in relation to an `:ExpressionStatement` which has been handed over by parameter *parent*. Parameters *srcName* and *trgName* are not predefined but are 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).

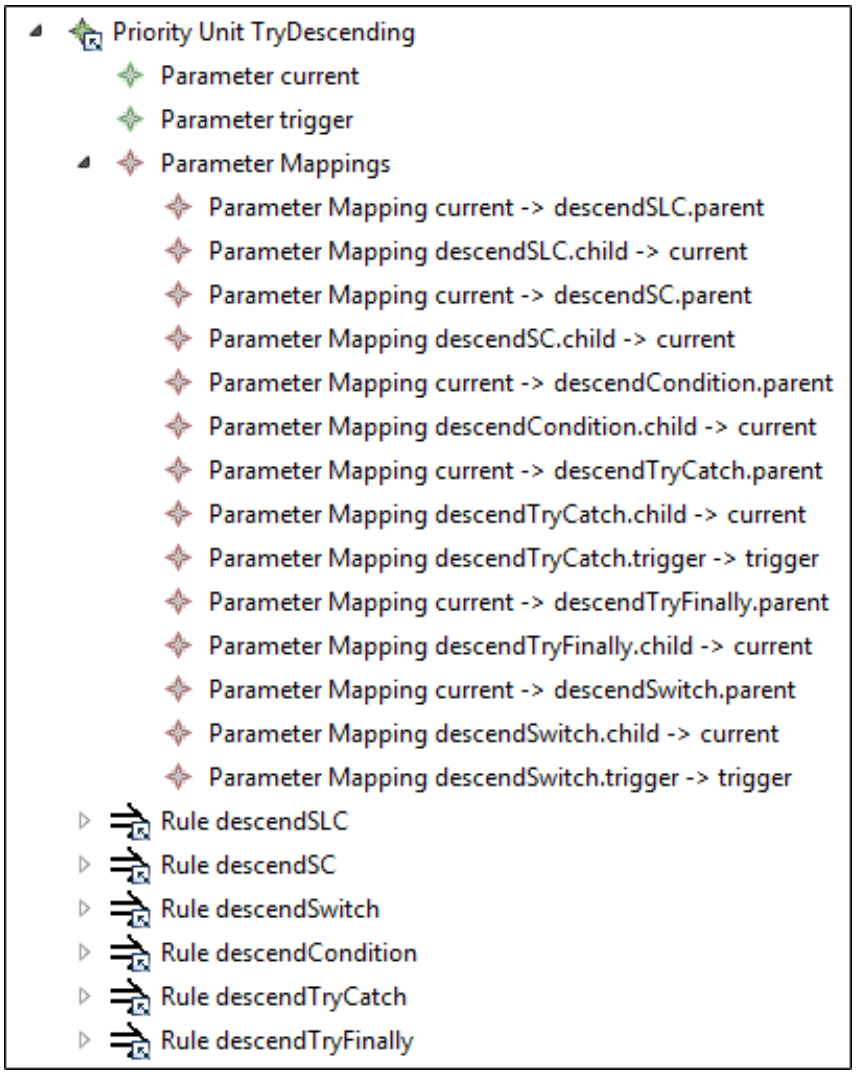

Figure 16: Priority unit *TryDescending* is part of unit *DecendLoop* (see Fig. 14) and performs the descending of the `:MethodClass` structure. Since HENSHIN does not support neither path expressions nor untyped nodes and untyped edges, each case has to be handled by a single rule separately. The priority 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.

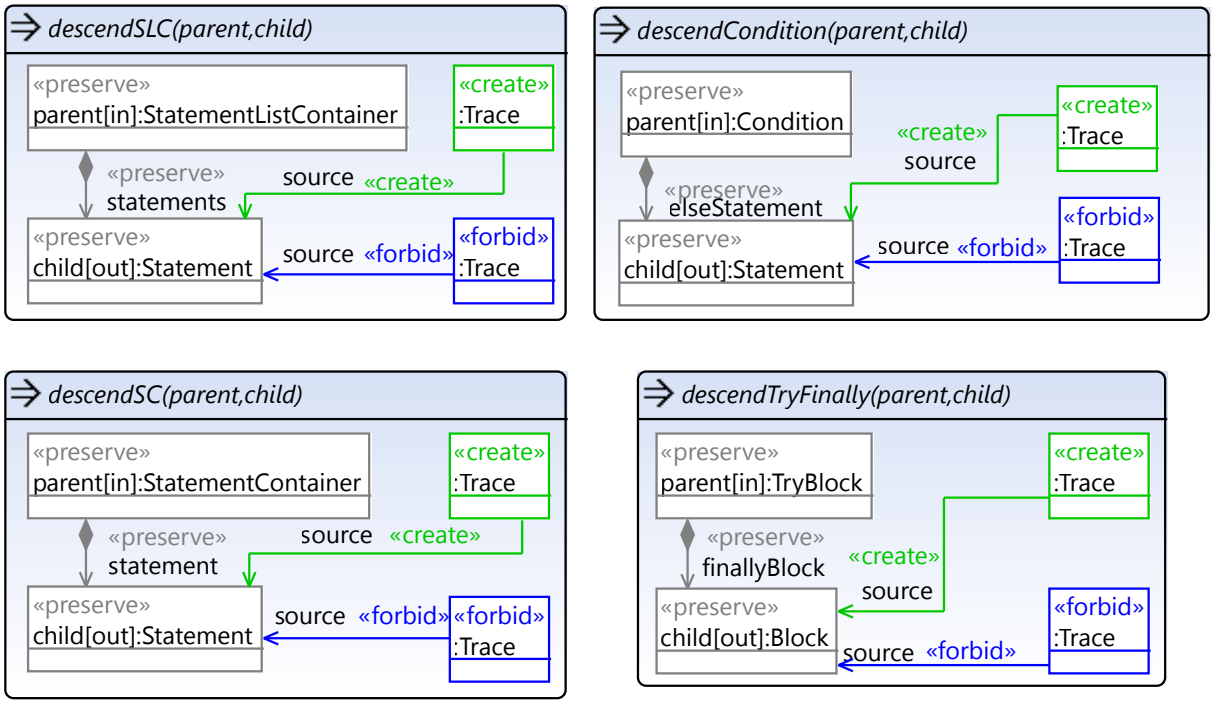

Figure 17: Rules *descendSLC*, *descendSC*, *descendCondition*, and *descendTryFinal* being part of the top-down traversal of the tree-like structure with `:ClassMethod` (being a subtype of `StatementListContainer`) as top-most element.

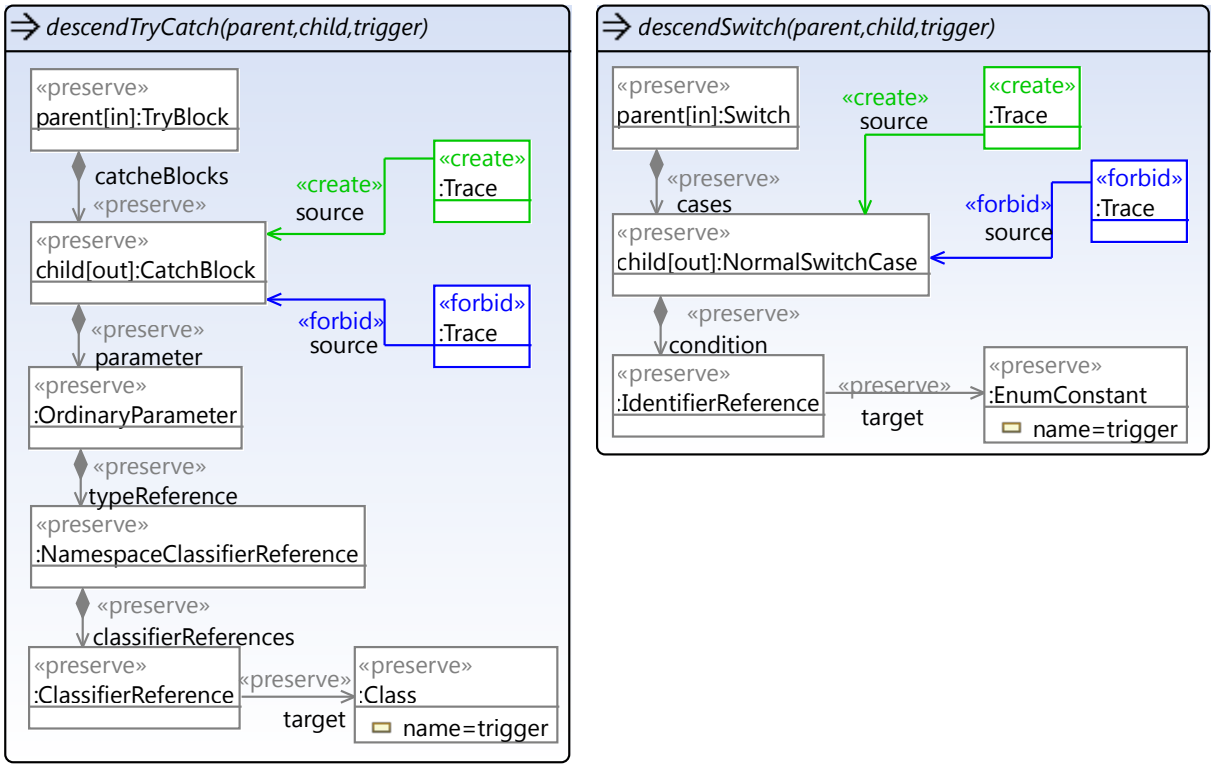

Figure 18: Rules *descendTryCatch* and *descendSwitch* performing a top-down traversal analog to the rules in Fig. 17. In addition, the trigger value is fetched to be used in the subsequent creation of a transition (see Fig.15).

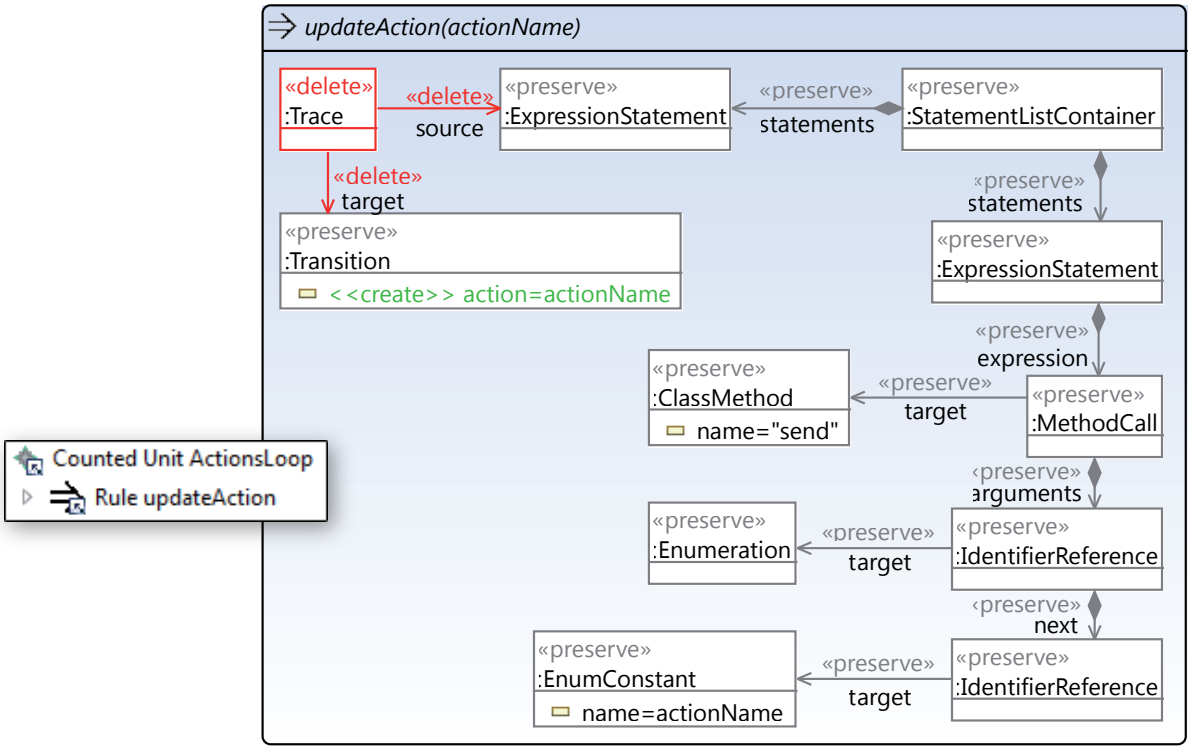

Figure 19: Counted unit *ActionsLoop* (left) and rule *updateAction* (right). As often as possible, the rule updates the `action` attribute value of any `:Transition` being associated by a `:Trace` object which points to an `:ExpressionStatement` and which in turn is contained by a `:StatementListContainer`. 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.

```
1  package de.jtietje.fh.ma.ttc2011;
2
3  import java.io.File;
4  import java.io.IOException;
5  import java.util.Collections;
6  import java.util.List;
7
8  import mapping.impl.MappingPackageImpl;
9
10 import org.eclipse.emf.common.util.EList;
11 import org.eclipse.emf.common.util.URI;
12 import org.eclipse.emf.compare.diff.metamodel.DiffElement;
13 import org.eclipse.emf.compare.diff.metamodel.DiffModel;
14 import org.eclipse.emf.compare.diff.service.DiffService;
15 import org.eclipse.emf.compare.match.metamodel.MatchModel;
16 import org.eclipse.emf.ecore.EObject;
17 import org.eclipse.emf.ecore.resource.Resource;
18 import org.eclipse.emf.ecore.resource.ResourceSet;
19 import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
20 import org.eclipse.emf.ecore.xmi.impl.EcoreResourceFactoryImpl;
21 import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;
22 import org.eclipse.emf.henshin.common.util.EmfGraph;
23 import org.eclipse.emf.henshin.interpreter.EmfEngine;
24 import org.eclipse.emf.henshin.interpreter.UnitApplication;
25 import org.eclipse.emf.henshin.model.TransformationSystem;
26 import org.eclipse.emf.henshin.model.TransformationUnit;
27 import org.eclipse.emf.henshin.model.impl.HenshinPackageImpl;
28 import org.eclipse.emf.henshin.model.resource.HenshinResourceFactory;
29 import org.eclipse.emf.henshin.trace.impl.TracePackageImpl;
30 import org.emftext.language.java.containers.CompilationUnit;
31 import org.emftext.language.java.containers.ContainersFactory;
32 import org.emftext.language.java.containers.Package;
33 import org.emftext.language.java.impl.JavaPackageImpl;
34
35 import statemachine.StateMachine;
36 import statemachine.impl.StatemachinePackageImpl;
37
38 /**
39  * M2M transformation with a JaMoPP Java model as source
40  * model and a state machine model as target. This is an
41  * implementation in terms of case study 1 of TTC2011
42  * (http://planet-research20.org/ttc2011/index.php?option=
43  * com_content &view=article&id=118&Itemid=160).
44  *
```

```
45  * @author Johannes Tietje
46  * @author Stefan Jurack
47  */
48 public final class JaMoPP2Statemachine {
49
50   /**
51    * The resource set for loading XMI files.
52    */
53   private static final ResourceSet RESOURCESET = new ResourceSetImpl();
54
55   /**
56    * time measurement purpose
57    */
58   private static long stopwatchValue = 0;
59
60   /**
61    * Program entry point.
62    *
63    * @param args
64    *          the command line arguments
65    */
66   public static void main(final String[] args) {
67     if (args.length != 1) {
68       System.err
69           .println("Usage: TCPStateExtract {1_small-model, 2_medium-model,
                3_big-model}");
70       System.exit(-1);
71     }// if
72
73     initializeFactories();
74     loadAndTransform(args[0]);
75   }// main
76
77   /**
78    * Loads and transforms the JaMoPP model with the given
79    * path.
80    *
81    * @param model
82    *          model to transform
83    */
84   private static void loadAndTransform(final String model) {
85     String modelFile = "model/source-models/" + model + ".xmi";
86     String henshinFile = "henshin/statemachine2.henshin";
87     String stateMachineXMIFile = "model/generated/" + model
88         + ".statemachine";
89
90     Package rootPackage = loadJaMoPPSourceModel(modelFile);
91
92     StateMachine stateMachine = performTransformation(
93         henshinFile, rootPackage);
94
95     saveModel(stateMachine, stateMachineXMIFile);
```

```
96
97      /**
98       * Compare result with reference model
99       */
100     String referenceFile = "model/statemachine/reference.statemachine";
101     compareEMF(referenceFile, stateMachineXMIFile);
102   }// loadAndTransform
103
104   /**
105    * Loads a given JaMoPP source code model. All contained
106    * {@link CompilationUnit}s are attached to a newly
107    * created {@link Package} as single root element. This is
108    * for convenience only.
109    *
110    * @param path
111    *          to the source code model
112    * @return a newly created {@link Package} with
113    *         compilation units contained.
114    */
115   private static Package loadJaMoPPSourceModel(final String path) {
116
117     JaMoPP2Statemachine.stopwatch("Start loading the model...");
118     Resource xmiResource = loadModel(path);
119     EList<EObject> resourceContents = xmiResource.getContents();
120
121     Package rootPackage = ContainersFactory.eINSTANCE
122         .createPackage();
123     List<CompilationUnit> compilationUnits = rootPackage
124         .getCompilationUnits();
125
126     for (EObject compilationUnit : resourceContents) {
127       if (compilationUnit instanceof CompilationUnit) {
128         compilationUnits.add((CompilationUnit) compilationUnit);
129       }// if
130     }// for
131
132     JaMoPP2Statemachine.stopwatch("Time to load the model: ");
133
134     return rootPackage;
135   }// loadJaMoPPSourceModel
136
137   /**
138    * Loads the henshin transformation model .
139    *
140    * @param path
141    *          the path of the Henshin file
142    * @param rootPackage
143    *          a {@link Package} containing a list of
144    *          compilation units
145    * @return a state machine object representing the result
146    *         of the transformations
147    */
```

```
148   private static StateMachine performTransformation(
149       final String path, final Package rootPackage) {
150
151     JaMoPP2Statemachine
152         .stopwatch("Start preparing the transformation ...");
153     Resource xmiResource = loadModel(path);
154     TransformationSystem transformationSystem = (TransformationSystem)
          xmiResource
155         .getContents().get(0);
156     TransformationUnit transformationUnit = transformationSystem
157         .findUnitByName("Start");
158     // internal representation of the EMF model
159     EmfGraph emfGraph = new EmfGraph();
160     emfGraph.addRoot(rootPackage);
161     EmfEngine emfEngine = new EmfEngine(emfGraph);
162     UnitApplication unitApplication = new UnitApplication(
163         emfEngine, transformationUnit);
164     JaMoPP2Statemachine.stopwatch("Time for preparations: ");
165
166     JaMoPP2Statemachine
167         .stopwatch("Start performing the transformation...");
168     if (unitApplication.execute()) {
169       System.out.println("Successful.");
170     } else {
171       System.out.println("Not successful.");
172     }// if else
173
174     JaMoPP2Statemachine.stopwatch("Time for transformation: ");
175
176     return (StateMachine) unitApplication
177         .getParameterValue("sm");
178   }// applyHenshinRules
179
180   /**
181    * Loads an EMF model file and returns it as a Resource.
182    *
183    * @param modelPath
184    *          the path to the model file
185    * @return the Resource representing the model file
186    */
187   private static Resource loadModel(final String modelPath) {
188     URI modelUri = URI.createFileURI(new File(modelPath)
189         .getAbsolutePath());
190     return RESOURCESET.getResource(modelUri, true);
191   }// loadModel
192
193   /**
194    * Serializes a given EMF model into an XMI file of the
195    * given path.
196    *
197    * @param eobject
198    *          the model to serialize
```

```
199     * @param path
200     *          the path of the resulting file
201     */
202    private static void saveModel(final EObject eobject,
203        final String path) {
204      URI ecoreUri = URI.createFileURI(new File(path)
205          .getAbsolutePath());
206
207      Resource ecoreResource = RESOURCESET
208          .createResource(ecoreUri);
209      ecoreResource.getContents().add(eobject);
210
211      try {
212        ecoreResource.save(null);
213      } catch (IOException e) {
214        e.printStackTrace();
215      }// try catch
216    }// saveStateMachine
217
218    /**
219     * Initializes related factories and packages.
220     */
221    private static void initializeFactories() {
222      Resource.Factory.Registry.INSTANCE
223          .getExtensionToFactoryMap().put("ecore",
224              new EcoreResourceFactoryImpl());
225      Resource.Factory.Registry.INSTANCE
226          .getExtensionToFactoryMap().put("xmi",
227              new XMIResourceFactoryImpl());
228      Resource.Factory.Registry.INSTANCE
229          .getExtensionToFactoryMap().put("henshin",
230              new HenshinResourceFactory());
231      Resource.Factory.Registry.INSTANCE
232          .getExtensionToFactoryMap().put("statemachine",
233              new XMIResourceFactoryImpl());
234
235      JavaPackageImpl.init();
236      HenshinPackageImpl.init();
237      StatemachinePackageImpl.init();
238      MappingPackageImpl.init();
239      TracePackageImpl.init();
240    }// initializeFactories
241
242    /**
243     * This method is for time measurement only and shall be
244     * ignored to understand the actual transformation
245     * process.
246     *
247     * @param message
248     */
249    private static final void stopwatch(final String message) {
250      if (stopwatchValue == 0) {
```

```
251       System.out.println(message);
252       stopwatchValue = System.nanoTime();
253     } else {
254       long temp = System.nanoTime();
255       System.out.format("%s %.4f sec \n", message,
256           ((float) (temp - stopwatchValue)) / 1000000000);
257       stopwatchValue = 0;
258     }// if else
259   }// stopwatch
260
261   /**
262    * Compares two given EMF files via EMF Compare. This
263    * method is only to check the result of the
264    * transformation, thus it is not part of the
265    * transformation process itself.
266    *
267    * @param referenceFile
268    *          the source model file
269    * @param compareFile
270    *          the target model file
271    */
272   private static void compareEMF(final String referenceFile,
273       final String compareFile) {
274     Resource leftResource = RESOURCESET.getResource(
275         URI.createFileURI(compareFile), true);
276     Resource rightResource = RESOURCESET.getResource(
277         URI.createFileURI(referenceFile), true);
278
279     MatchModel match = null;
280
281     try {
282       match = (new StatemachineMatcher()).resourceMatch(
283           leftResource, rightResource,
284           Collections.<String, Object> emptyMap());
285     } catch (InterruptedException e) {
286       e.printStackTrace();
287     }// try catch
288
289     DiffModel diff = DiffService.doDiff(match, false);
290
291     System.out.println("Printing the differences...");
292
293     for (DiffElement diffElement : diff.getDifferences()) {
294       System.out.println(diffElement.toString());
295     }// for
296   }// compareEMF
297
298 }// class
```

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

```
1  package de.jtietje.fh.ma.ttc2011;
2
3  import java.util.List;
4
5  import org.eclipse.emf.compare.FactoryException;
6  import org.eclipse.emf.compare.match.engine.GenericMatchEngine;
7  import org.eclipse.emf.ecore.EObject;
8
9  import statemachine.State;
10 import statemachine.Transition;
11
12 /**
13  * A custom matcher for statemachine model elements as
14  * helper class for EMF Compare. This is used to compare the
15  * result of the JaMoPP2Statemachine transformation with an
16  * exemplar given by the TTC host.
17  *
18  * @author Johannes Tietje
19  */
20 public class StatemachineMatcher extends GenericMatchEngine {
21
22   @Override
23   protected final EObject findMostSimilar(final EObject eObj,
24       final List<EObject> list) throws FactoryException {
25     if (eObj instanceof Transition) {
26       Transition source = (Transition) eObj;
27
28       for (EObject eObject : list) {
29         if (eObject instanceof Transition) {
30           Transition potentialMostSimilar = (Transition) eObject;
31           boolean isMostSimilar = isSimilar(source.getSrc(),
32               potentialMostSimilar.getSrc())
33               && isSimilar(source.getDst(),
34                   potentialMostSimilar.getDst());
35
36           if (isMostSimilar) {
37             return potentialMostSimilar;
38           }
39         }
40       }
41     }
42
43     return super.findMostSimilar(eObj, list);
44   }
45
46   @Override
47   protected final boolean isSimilar(final EObject obj1,
48       final EObject obj2) throws FactoryException {
49     if (obj1 instanceof State || obj2 instanceof State) {
50       State firstState = (State) obj1;
```

```
51        State secondState = (State) obj2;
52
53        return firstState.getName().equals(secondState.getName());
54      } else {
55        if (obj1 instanceof Transition
56            || obj2 instanceof Transition) {
57          Transition firstTransition = (Transition) obj1;
58          Transition secondTransition = (Transition) obj2;
59
60          boolean preCheck = firstTransition.getAction().equals(
61              secondTransition.getAction())
62              && firstTransition.getTrigger().equals(
63                  secondTransition.getTrigger());
64
65          if (preCheck) {
66            return isSimilar(firstTransition.getDst(),
67                secondTransition.getDst())
68                && isSimilar(firstTransition.getSrc(),
69                    secondTransition.getSrc());
70          } else {
71            return false;
72          }
73        }
74      }
75
76      return super.isSimilar(obj1, obj2);
77    }
78  }
```