Transformation Tool Contest 2010
1-2 July 2010, Malaga, Spain

Steffen Mazanek
Arend Rensink
Pieter Van Gorp

(page intentionally left blank)

Contents

Model Migration case study **1**
Model MigrationWith GReTL 7
Migrating Activity Diagrams with Epsilon Flock 30
Model Migration with MOLA 38
A GrGen.NET solution of the Model Migration Case for the Transformation Tool Contest 2010 61
Migrating UML Activity Models with COPE 72
UML Model Migration with PETE 85
Abstract and Concrete Syntax Migration of Instance Models 100
A Graph Transformation Case Study for the Topology Analysis of Dynamic Communication Systems **107**
Solving the Topology Analysis Case Study with GROOVE 119
Abstract topology analysis of the join phase of the merge protocol 127
Topology Analysis of Car Platoons Merge with FujabaRT & TimedStoryCharts - a Case Study 134
Ecore to GenModel Case Study **149**
Modeling the “Ecore to GenModel” Transformation with EMF Henshin 153
ECore2GenModel with Mitra and GEF3D 166
Ecore to Genmodel case study solution using the Viatra2 framework 187

(page intentionally left blank)

Model Migration Case for TTC 2010

Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science, University of York, UK.
[louis,dkolovos,paige,fiona]@cs.york.ac.uk

Abstract. Using an example from the Unified Modelling Language, we invite submissions to explore the ways in which model transformation languages can be used to update models in response to metamodel changes.

1 Introduction

MDE introduces additional challenges for controlling and managing software evolution [8]. For example, when a metamodel evolves, instance models might no longer conform to the structures and rules defined by the metamodel. When an instance model does not conform to its metamodel, it cannot be manipulated with metamodel-specific editors, cannot be managed with model management operations and, in some cases, cannot be loaded with modelling tools. *Model migration* is a development activity in which instance models are updated to re-establish conformance in response to metamodel evolution.

Several approaches to automating model migration have been proposed. Sprinkle et al. [12] were the first to describe co-evolution as distinct from the more general activity of model-to-model transformation. Presently, various languages are used for specifying model migration, such as the Atlas Transformation Language (ATL) [6] in work by Cicchetti et al. [1], and the general-purpose programming language Groovy [7] in COPE [5]. There is little work, however, that compares the languages used for specifying model migration.

To explore and compare ways in which model migration can be specified, we propose a case from the evolution of the UML. The way in which activity diagrams are modelled in the UML has changed significantly between versions 1.4 and 2.1 of the specification. In the former, activities were defined as a special case of state machines, while in the latter they are defined atop a more general semantic base[1] [11].

2 Activity Diagrams in UML

Activity diagrams are used for modelling lower-level behaviours, emphasising sequencing and co-ordination conditions. They are used to model business processes and logic [10]. Figure 1 shows an activity diagram for filling orders. The

[1] A variant of generalised coloured Petri nets.

diagrams is partitioned into three *swimlanes*, representing different organisational units. *Activities* are represented with rounded rectangles and *transitions* with directed arrows. *Fork* and *join* nodes are specified using a solid black rectangle. *Decision* nodes are represented with a diamond. Guards on transitions are specified using square brackets. For example, in Figure 1 the transition to the restock activity is guarded by the condition `[not in stock]`. Text on transitions that is not enclosed in square brackets represents a trigger event. In Figure 1, the transition from the restock activity occurs on receipt of the asynchronous signal called `receive stock`. Finally, the transitions between activities might involve interaction with objects. In Figure 1, the Fill Order activity leads to an interaction with an object called `Filled Object`.

Between versions 1.4 and 2.2 of the UML specification, the metamodel for activity diagrams has changed significantly. The sequel summarises most of the changes. For full details, refer to [9] and [10].

3 Evolution of Activity Diagrams

Figures 2 and 3 are simplifications of the activity diagram metamodels from versions 1.4 and 2.2 of the UML specification, respectively. In the interest of clarity, some features and abstract classes have been removed from Figures 2 and 3.

Some differences between Figures 2 and 3 are: activities have been changed such that they comprise nodes and edges, actions replace states in UML 2.2, and the subtypes of control node replace pseudostates. For full details of the changes made between UML 1.4 and 2.2, refer to [9] and [10].

The model to be migrated is shown in Figure 1, and is based on [9, pg3-165]. A migrating transformation should migrate the activity diagram shown in Figure 1 from UML 1.4 to UML 2.2. The UML 1.4 model, the migrated, UML 2.2 model, and the UML 1.4 and 2.2 metamodels are available from[2].

Submissions will be evaluated based on the following three criteria:

- **Correctness**: Does the transformation produce a model equivalent to the migrated UML 2.2. model included in the case resources?
- **Conciseness**: How much code is required to specify the transformation? (In [12] et al. propose that the amount of effort required to codify migration should be directly proportional to the number of changes between original and evolved metamodel).
- **Clarity**: How easy is it to read and understand the transformation? (For example, is a well-known or standardised language?)

Submissions might also consider the three extensions discussed below, in Sections 3.1, 3.2 and 3.3.

[2] `http://www.cs.york.ac.uk/~louis/ttc/`

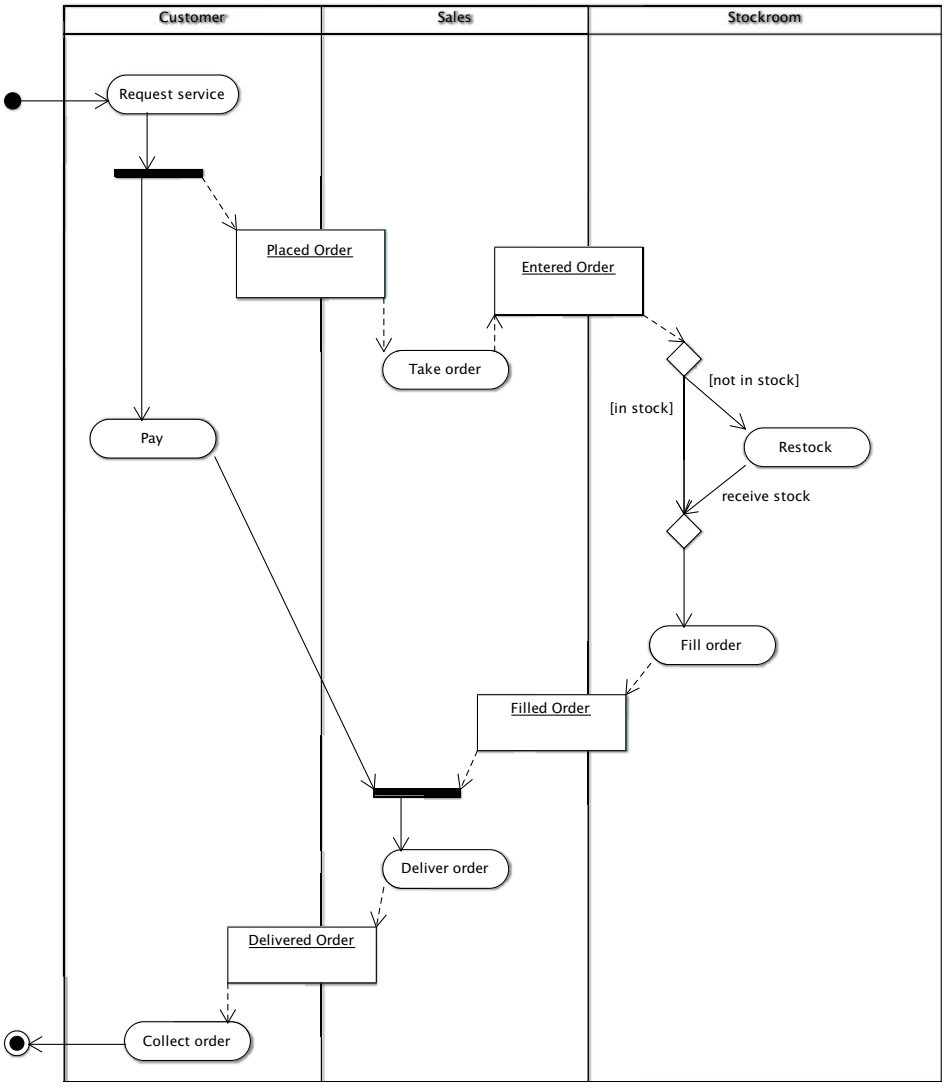


Fig. 1. Activity model to be migrated.

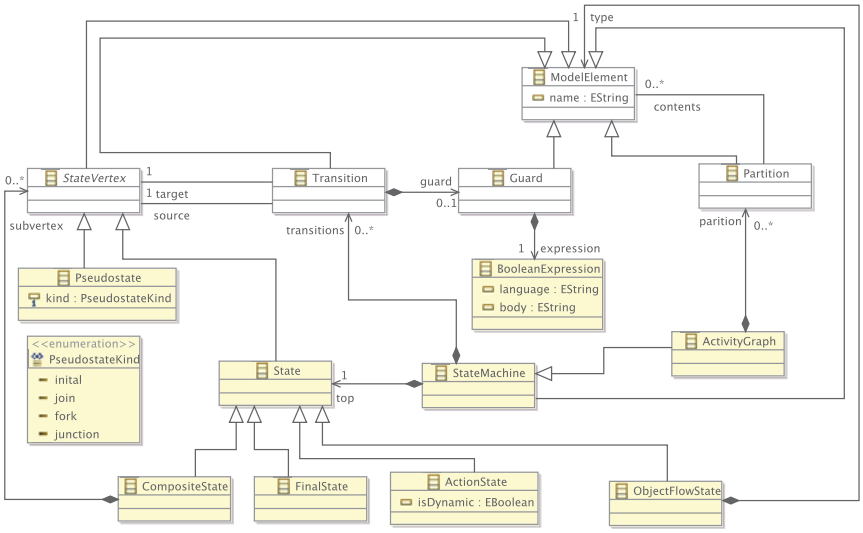


Fig. 2. UML 1.4 Activity Graphs (based on [9]).

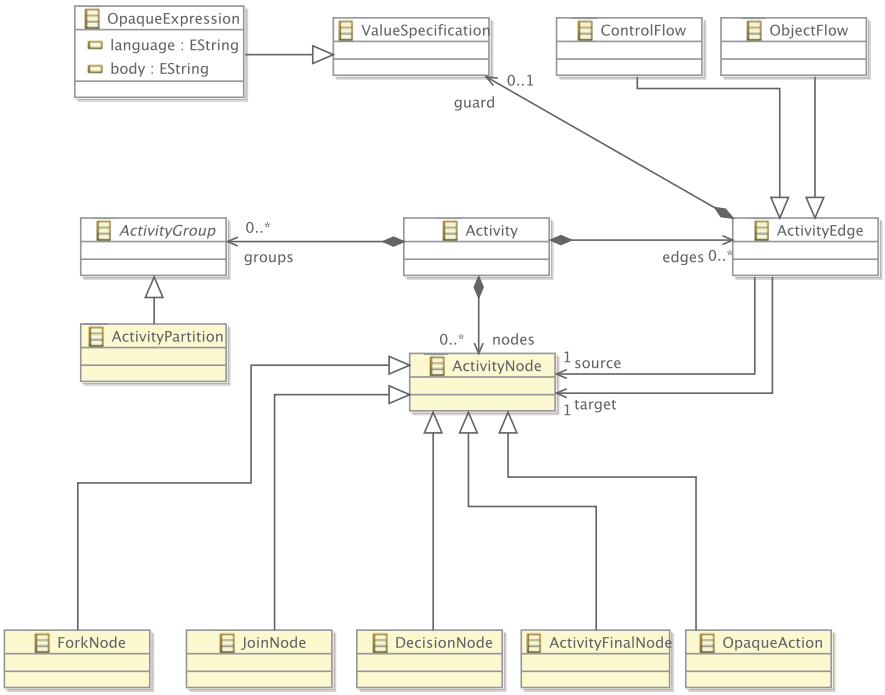


Fig. 3. UML 2.2 Activity Diagrams (based on [10]).

3.1 Alternative Object Flow State Migration Semantics

Following the submission of this case, much discussion on the TTC forums[3] revealed an ambiguity in the UML 2.2 specification. Consequently, the migration semantics for the ObjectFlowState UML 1.4 concept are not clear from the UML 2.2. specification.

In the core task described above, instances of ObjectFlowState should be migrated to instances of ObjectNode. Any instances of Transition that have an ObjectFlowState as their source or target should be migrated to instances of ObjectFlow. Listing 1.1 shows an example application of this migration semantics. The top line of Listing 1.1 shows instances of UML 1.4 metaclasses, include an instance of ObjectFlowState. The bottom line of Listing 1.1 shows the equivalent UML 2.2. instances according to this migration semantics. Note that the Transitions, t1 and t2, is migrated to an instance of ObjectFlow. Likewise, the instance of ObjectFlowState, s2, is migrated to an instance of ObjectFlow.

```
s1:State <- t1:Transition -> s2:ObjectFlowState <- t2:Transition -> s3:State

s1:ActivityNode <- t1:ObjectFlow -> s2:ObjectNode <- t2:ObjectFlow -> s3:
    ActivityNode
```

Listing 1.1. Migrating Actions

This extension consider an alternative migration semantics for ObjectFlowState. For this extension, instances of ObjectFlowState (and any connected Transitions) should be migrated to instances ObjectFlow, as shown by the example in Listing 1.2. Note that the UML 2.2 ObjectFlow, f1, replaces t1, t2 and s2.

```
s1:State <- t1:Transition -> s2:ObjectFlowState <- t2:Transition -> s3:State

s1:ActivityNode <- f1:ObjectFlow -> s3:ActivityNode
```

Listing 1.2. Migrating Actions

3.2 Concrete Syntax

The UML specifications provide no formally defined metamodel for the concrete syntax of UML diagrams. However, some UML tools store diagrammatic information in a structured manner using XML or a modelling tool. For example, the Eclipse UML 2 tools [4] store diagrams as GMF [3] diagram models. As such, submissions might explore the feasibility of migrating the concrete syntax of the activity diagram shown in Figure 1 to the concrete syntax in their chosen UML 2 tool. To facilitate this, the case resources include an ArgoUML [2] project containing the activity diagram shown in Figure 1.

[3] `http://planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewgroup&groupid=4&Itemid=150` (registration required)

3.3 XMI

Because XMI has evolved at the same time as UML, UML 1.4 tools most likely produce XMI of a different version to UML 2.2 tools. For instance, ArgoUML produces XMI 1.2 for UML 1.4 models, while the Eclipse UML2 tools produce XMI 2.1 for UML 2.2. As an extension to the case outline above, submissions might consider how to migrate a UML 1.4 model represented in XMI 1.x to a UML 2.1. model represented in XMI 2.x. To facilitate this, the UML 1.4 model shown in Figure 1 is available in XMI 1.2 as part of the case resources.

Acknowledgement. The work in this paper was supported by the European Commission via the MADES project, co-funded by the European Commission under the "Information Society Technologies" Seventh Framework Programme (2009-2012).

References

1. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in MDE. In *Proc. EDOC*, pages 222–231. IEEE Computer Society, 2008.
2. CollabNet. ArgoUML [online]. [Accessed 5 March 2010] Available at: `http://argouml.tigris.org/`, 2008.
3. Eclipse. Graphical Modelling Framework project [online]. [Accessed 19 September 2008] Available at: `http://www.eclipse.org/modeling/gmf/`, 2008.
4. Eclipse. UML2 Model Development Tools project [online]. [Accessed 5 March 2010] Available at: `http://www.eclipse.org/modeling/mdt/uml2`, 2009.
5. M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - automating coupled evolution of metamodels and models. In *Proc. ECOOP*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.
6. F. Jouault and I. Kurtev. Transforming models with ATL. In *Proc. Satellite Events at MoDELS*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.
7. Glover A. King P. Laforge G. Koenig, D. and J. Skeet. *Groovy in Action*. Manning Publications, Greenwich, CT, USA, 2007.
8. T. Mens and S. Demeyer. *Software Evolution*. Springer-Verlag, 2007.
9. OMG. Unified Modelling Language 1.4 Specification [online]. [Accessed 5 March 2010] Available at: `http://www.omg.org/spec/UML/1.4/`, 2001.
10. OMG. Unified Modelling Language 2.2 Specification [online]. [Accessed 5 March 2010] Available at: `http://www.omg.org/spec/UML/2.2/`, 2007.
11. Bran Selic. Whats new in UML 2.0? *IBM Rational software*, 2005.
12. Jonathan Sprinkle and Gábor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15(3-4):291–307, 2004.

Model Migration With GReTL

Tassilo Horn
horn@uni-koblenz.de
Institute for Software Technology
University Koblenz-Landau

May 10, 2010

Abstract

This paper briefly introduces the GReTL transformation language by presenting a solution to the TTC 2010's Model Migration case study, which handles both the core as well as the object flow extension task.

1 Introduction

The GReTL transformation language is implemented on the foundations of a technological space [7] called the *TGraph approach* [3]. In this approach, models are represented as TGraphs: typed, attributed, ordered, directed graphs. Edges are not only references but first-class objects that have a type, can have attributes and can always be traversed in both directions. The Java library *JGraLab*[1] implements the framework of that approach. For a more detailed introduction, have a look at appendix A.

The transformation language *GReTL* (*Graph Repository Transformation Language*, [6]) is a Java framework for programming transformations on TGraphs. A transformation is a Java *strategy object* [5] that *operationally* transforms a given model using a small set of operations provided by the GReTL API. A GReTL transformation constructs the target metamodel (a *schema*, consisting of *vertex* and *edge classes* with *attributes*) programatically[2], and thereby it *declaratively* specifies how to migrate the source model elements into the newly constructed schema using the *GReQL* query language [4] (see appendix A.1).

Instead of specifying rules relating source and target metamodel elements, GReTL uses a mathematical, set-oriented approach. For each type created in the target schema, a GReQL query is provided and evaluated on the source graph, which calculates a set of arbitrary *archetypes*. For each member in this *archetype set*, a new element of that new type is instantiated in the target graph. Those new elements are called the *images* of their *archetypes*, and the traceability information is saved. Note that archetypes can be arbitrary objects: source model vertices or edges, strings, numbers, or tuples, sets, maps and lists thereof.

[1] http://jgralab.uni-koblenz.de

[2] It is also possible to transform to an existing target schema, but this is only a special case.

2 The Case Study's Transformation

In the following, GReTL is explained using parts of the Activity1ToActivity2 migration transformation solving the Model Migration case's **core** and **object flow extension tasks**. This transformation uses the UML 1.4 activity diagram shown in figure 1 as input.

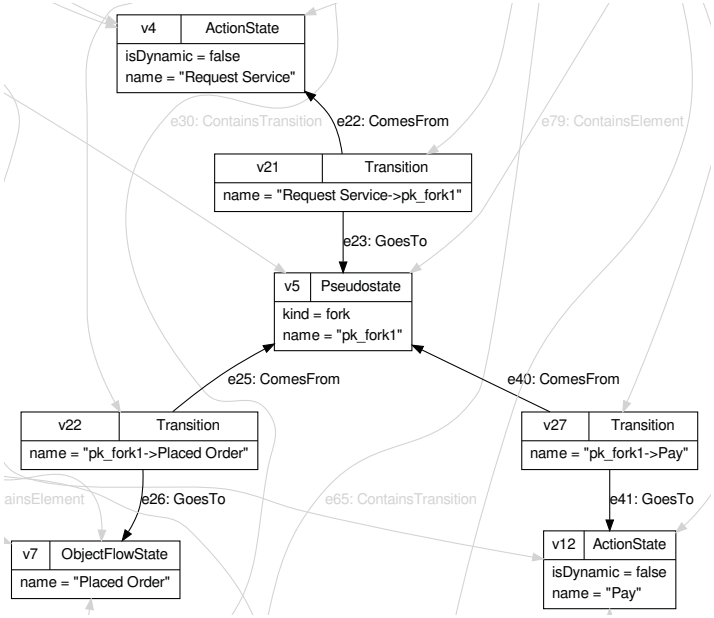


Figure 1: A visualization of a small part of the UML 1.4 source model

The schema the source model conforms to is shown in figure 2.

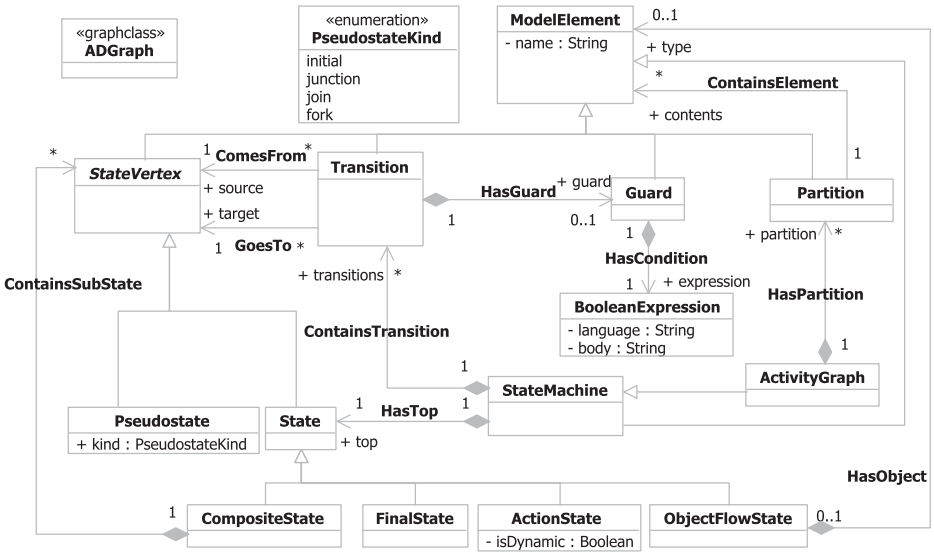


Figure 2: The UML 1.4 activity diagram source schema

The Activity1ToActivity2 transformation creates the UML 2.2 activity diagram schema depicted in figure 3, and it migrates the UML 1.4 model from figure 1 to an instance of the new schema.

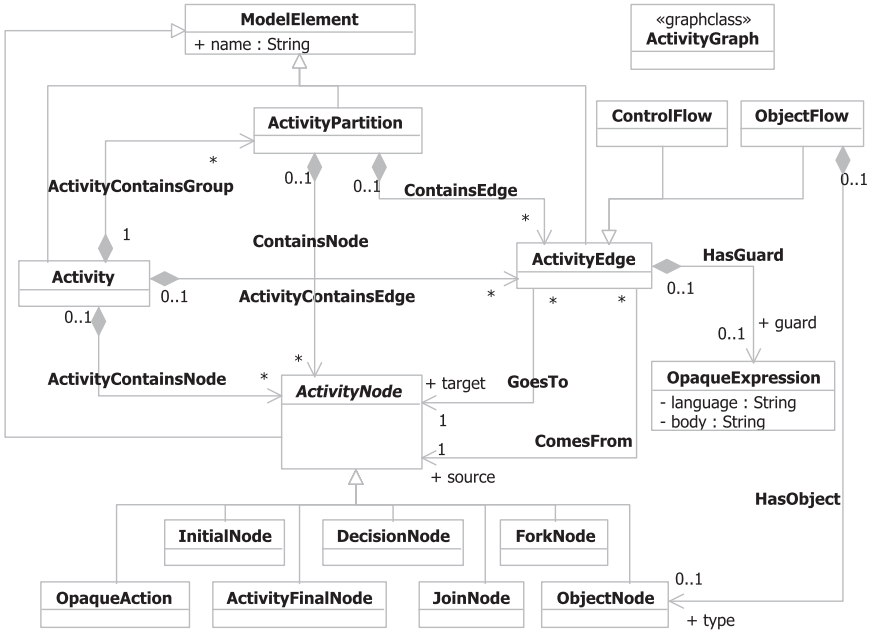


Figure 3: The target UML 2.2 activity diagram schema the tranformation constructs

All GReTL transformation have a common template, which is presented in the following listing.

```
public class Activity1ToActivity2 extends Transformation {
    @Override
    protected void transform() {
        // Here go all calls to transformation operations...
    }
}
```

The Activity1ToActivity2 transformation extends the abstract Transformation class provided by the framework, and it overrides its transform() method. Inside that, calls to the basic transformation operations inherited from Transformation are placed, which realize the transformation's behavior. Those are presented in the following.

2.1 Creating Vertex Classes and Vertices

To create a vertex class in the target schema and to create vertices in the target graph, the Transformation class provides the following two methods.

```
protected final VertexClass createAbstractVertexClass(String qName)
protected final VertexClass createVertexClass(String qName, String semanticExpression)
```

The first method is used for creating an abstract vertex class in the target schema. Because an abstract class cannot have instances, only the qualified name has to be given.

The second method creates a concrete vertex class with the given qualified name. The second parameter semanticExpression specifies the instances of the newly created vertex type

that have to be created in the target graph. It is a GReQL query, which is evaluated on the source graph and has to result in a set. For each member of this set, a new vertex of type qName is created in the target graph. The mappings from members of this set (*archetypes*) to target graph vertices created in response (*images*) is saved as a function img_{qName}. The inverse function $arch_{qName}$ is also saved for performance reasons. The GReTL framework makes both functions accessible in following semantic expressions and enforces their bijectivity in order to allow for bidirectional navigation between images and archetypes. These functions are used later on when creating edges, which need to refer the vertices they start and end at and when creating attributes.

Turning to the Activity1ToActivity2 transformation, here is an operation invocation to create the vertex class InitialNode in the target schema and to instantiate InitialNode vertices.

```
createVertexClass("InitialNode",
    "from ps : V{Pseudostate} with ps.kind = \"initial\" reportSet ps end");
```

The semantic expression results the set of all source graph Pseudostate vertices, that have their kind attribute set to the enum literal initial. For each of those pseudostates, a new InitialNode vertex is created in the target graph, and the mappings from source model pseudostates to target model initial nodes is saved in a function $img_{InitialNode}$ (and its reverse $arch_{InitialNode}$).

Other pseudostates have to be mapped to other target model vertices of different types. Because they only vary in the value of their kind attribute, this can be factored out in a simple loop over an array of target schema type names and source model kind values. The fact that GReTL transformations are POJOs comes in handy here.

```
for (String[] s : new String[][] { { "InitialNode", "initial" }, { "ForkNode", "fork" },
                    { "JoinNode", "join" }, { "DecisionNode", "junction" } }) {
    createVertexClass(s[0],
        "from ps : V{Pseudostate} with ps.kind = \"" + s[1] + "\" reportSet ps end"); }
```

So these few lines create the vertex classes Initial-, Fork-, Join- and DecisionNode in the target schema. On the instance level, they populate the target graph with new instances of these four classes, one instance per source model pseudostate of a given kind. The mappings are saved in four corresponding *image* (and *archetype*) functions.

There are several other source metamodel vertex classes which have a one-to-one relationship with target metamodel vertex classes.

```
for (String[] s : new String[][] {              { "Activity",          "ActivityGraph" },
        { "ActivityPartition", "Partition" },   { "ActivityFinalNode", "FinalState" },
        { "OpaqueAction",      "ActionState" }, { "ObjectNode",        "ObjectFlowState" },
        { "OpaqueExpression",  "Guard" } }) {
    createVertexClass(s[0], "V{" + s[1] + "}"); }
```

Again, these "renames" can be easily implemented by iterating over the elements of an array, which contains pairs of the form (*NewVertexClass*, *OldVertexClass*). The semantic expression V{*OldVertexClass*} returns the set of all source model vertices of type *OldVertexClass*, and for each member in this set, a target graph vertex of type *NewVertexClass* is created in the target graph. Six image and archetype functions providing bidirectional navigation between old and new vertices are created implicitly.

2.2 Creating Edge Classes and Edges

To create an edge class and edge instances, the following operations are provided by the Transformation class.

```
protected final EdgeClass createAbstractEdgeClass(String qName,
    IncidenceClassSpec fromSpec, IncidenceClassSpec toSpec)
protected final EdgeClass createEdgeClass(String qName,
    IncidenceClassSpec fromSpec, IncidenceClassSpec toSpec, String semanticExpression)
```

The class IncidenceClassSpec is only a convenience wrapper encapsulating the properties of an edge class end like the connected vertex class, multiplicities, a role name and the aggregation kind. Any property except the connected vertex class is optional, and appropriate default values are used for the omitted ones.

Because abstract edge classes cannot be instatiated, there is no semantic expression in the createAbstractEdgeClass() signature. But for concrete edge classes, this parameter is again a GReQL query, which is evaluated on the source model. In contrast to vertices, an edge cannot be created on its own, but a start and an end vertex have to be provided. Therefore, the semantic expression has to evaluate to a set of triples. The first element in each triple defines the archetype of the new edge. The second element specifies the archetype of the desired start vertex in the target graph. The third element specifies the desired end vertex archetype. Again, the image and archetype functions are exported as img_{qName} and $arch_{qName}$.

Turning to the Activity1ToActivity2 transformation, here is the operation invocation to create the ActivityContainsGroup edge class in the target schema and to populate the target graph with instances assigning ActivityPartition vertices to the Activity vertex containing this partition.

```
1 createEdgeClass("ActivityContainsGroup",
2     new IncidenceClassSpec(vc("Activity")),
3     new IncidenceClassSpec(vc("ActivityPartition"), AggregationKind.COMPOSITE),
4     "from e : E{HasPartition} reportSet e, startVertex(e), endVertex(e) end");
```

The incidence class specs specify that this edge type starts at the vertex class Activity and ends at the vertex class ActivityPartition. It has composition semantics, where Activity is the whole and ActivityPartition is the part (see figure 3).

The semantic expression specifies a set of triples. There is one triple for each source model HasPartition edge, and because these edges are the first component of each triple, they are the archetypes for the new ActivityContainsGroup edges in the target model. The second and third compontent of each result set triple specify the archetypes of the start and end vertex in the target graph. The new ActivityContainsGroup edges's start and end vertices are exactly the images of the source and target vertices of the source model HasPartition edges, i.e. images of ActivityGraph and Partition. Those were already transformed to Activities and ActivityPartitions in the last operation call of section 2.1.

2.3 Adding Type Hierarchies

Till now, the specialization relationships in the target schema have not been established. The Transformation class provides the following two methods for this purpose.

```
protected final void addSubClasses(VertexClass superClass, VertexClass... subClasses)
protected final void addSubClasses(EdgeClass superClass, EdgeClass... subClasses)
```

Both make the vertex or edge classes given as second to last parameters specializations of the vertex or edge class given as first parameter. Calling these methods has no direct effect

on the instance level. But there is an effect on the image and archetype functions of the super vertex or edge class. After estabishing a specialization, the image and archetype functions of the superclass contain all former mappings plus all the mappings of the given subclasses' image and archetype functions. Therefore, in any type hierarchy, the archetypes have to be disjoint in order to ensure bijectivity.

From the Activity1ToActivity2 transformation, one example invocation is presented here.

```
1 addSubClasses(activityNode, vc("OpaqueAction"), vc("InitialNode"),
2   vc("ActivityFinalNode"), vc("DecisionNode"), vc("JoinNode"),
3   vc("ForkNode"), vc("ObjectNode"));
```

The vertex classes OpaqueAction, InitialNode, ActivityFinalNode, DecisionNode, JoinNode, ForkNode and ObjectNode are made subclasses of the vertex class ActivityNode, which is referenced the variable activityNode here. As a result, the image and archetype functions $img_{ActivityNode}$ and $arch_{ActivityNode}$ contain all mappings of the corresponding subclass functions.

2.4 Creating Attributes and Setting Attribute Values

The following listing shows the method for creating an attribute and setting its values.

```
protected final Attribute createAttribute(AttributeSpec attrSpec, String semanticExpression)
```

The class AttributeSpec encapsulates the class the attribute belongs to, its name and domain and an optional default value.

The semantic expression is again a GReQL query, but here it has to evaluate to a map, which maps attrElemClass-archetypes to the value that should be set for their target graph images.

The following operation call of the Activity1ToActivity2 transformation creates the name attribute of the UML 2.2 ModelElement class, and it sets the value for all target ModelElement vertices excluding ActivityEdges.

```
1 createAttribute(new AttributeSpec(modelElement, "name", getStringDomain()),
2     "from me : difference(keySet(img_ModelElement), keySet(img_ActivityEdge)) "
3   + "reportMap me, me.name end                                                 ");
```

The given attribute spec specifies the schema properties. The attribute belongs to the vertex class ModelElement referenced by a variable. Its name is name, and its domain is String.

The semantic expression specifies a function (reportMap). All source model ModelElements, which are the archetyes of target graph ModelElements excluding the archetypes of ActivityEdges are iterated[3]. The keys are the archetypes, and the values are the values of their name attribute. So the operation call basically copies the old model elements' name values over to the corresponding target graph model elements, but it skips the setting of the values for target model ActivityEdges, which are handled separately by the transformation.

2.5 Variation Between Core and Object Flow Extension Task

In this section, the variation in the transformation needed for creating a target UML 2.2 activity diagram according to the *core* or the *object flow extension task* are discussed. The variation is about how source model ObjectFlowStates are transformed into the target model.

A visualization of such an object flow state in the source model is given in figure 4.

[3]The GReQL function keySet() returns the set of keys of a map, that is the domain of the $img_{ModelElement}$ function. In GReQL all map access functions are named according to the method names of the java.util.Map interface.

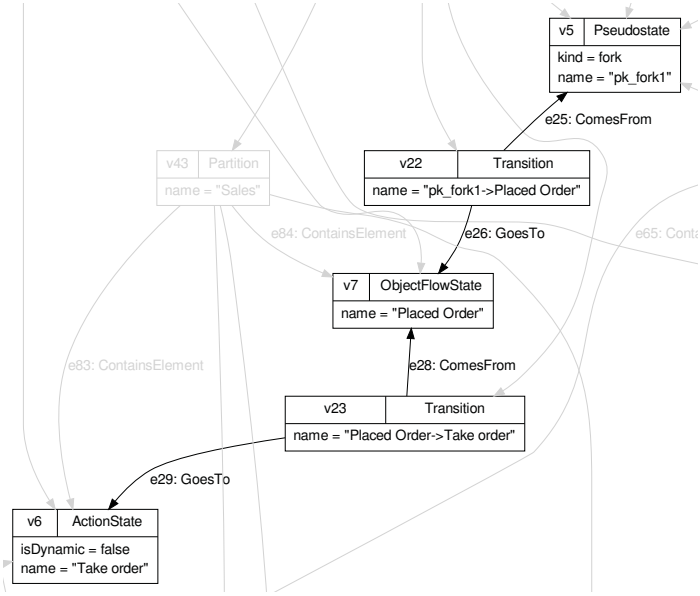


Figure 4: An source model object flow state and its surrounding

From the fork pseudostate v5, an object named "Placed Order" represented by the ObjectFlowState v7 is transfered to the ActionState v6 via the two Transition vertices v22 and v23 and their connecting ComesFrom (e25, e28) and GoesTo (e26, e29) edges. In the core task, the transformed model is pretty isomorphic but uses different types, whereas in the object flow extension task, the structure is quite different.

The variation is placed into a simple Java switch statement, which dispatches according to the value of a field task, which can be set with a usual setter method. First, the *preceeding*

```
1  // preceeding shared part...
2  switch (task) {
3      case CORE:
4          // core part...
5          break;
6      case OBJECT_FLOW_EXTENSION:
7          // object flow extension part...
8          break;
9      default: throw new GReTLException(context, "Unknown task '" + task + "'!");
10 }
11 // following shared part...
```

shared part is discussed. Thereafter, the *object flow extension part* is explained. Due to its higher complexity, an appropriate selection of archetypes for the varying target graph elements is done here. Then, the *core part* is discussed which obeys the archetype selection schema of the extension part. Finally, the *following shared part* is discussed, which is again completely identical for both tasks and gets along without any distinction.

Preceeding Shared Part. The metamodel of both tasks is identical, so the creation of metamodel elements whose instances are affected by the variation can be done in the preceeding shared part of the transformation, which also includes the operations already presented.

```
1 VertexClass objectFlow = createVertexClass("ObjectFlow");
2 EdgeClass hasObject = createEdgeClass("HasObject",
3     new IncidenceClassSpec(objectFlow, 0, 1),
4     new IncidenceClassSpec(vc("ObjectNode"), 0, 1, AggregationKind.COMPOSITE));
```

These two operation invocations create the ObjectFlow vertex class and the HasObject composition edge class according to the target schema (figure 3). But both of them don't specify a semantic expression, and no instances of these types are created in the target model at that time.

Object Flow Extension Task. The target graph snippet corresponding to the source part of figure 4 when transformed according to the object flow extension task is shown in figure 5. The source fork pseudostate v5 was transformed to the ForkNode v20, and the action state v6 has become the OpaqueAction v7. Also, the structure is not isomorphic to the source model. The transferred "Placed Order" object is represented by the ObjectNode v13, but it is not source and target of two individual ObjectFlows. Instead, it is connected to one single ObjectFlow v24 with an HasObject edge, and the single object flow directly leads from the ForkNode v20 to the OpaqueAction v7.

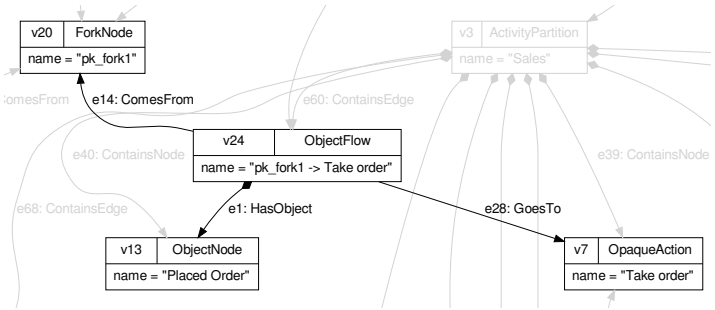


Figure 5: The source model part of figure 4 transformed according to the **object flow extension task**

To create this target graph structure, instances of the vertex class ObjectFlow and the edge class HasObject have to be created. Both classes are already created in the schema and only need to be instantiated. Therefore, the Transformation class provides the operations instantiateVertices() and instantiateEdges(), which only work on the instance level[4].

```
1 instantiateVertices(objectFlow,
2     "from t1 : V{Transition}, t2 : V{Transition}                "
3   + "with t1 -->{GoesTo} & {ObjectFlowState} <--{ComesFrom} t2 "
4   + "reportSet t1, t2 end                                      ");
5
6 instantiateEdges(hasObject,
7     "from t : keySet(img_ObjectFlow)                    "
8   + "reportSet t, t, theElement(t[0] -->{GoesTo}) end ");
```

[4]If the transformation used an existing target schema instead of creating it programmatically, only these operations would be used.

One target model ObjectFlow vertex is created for any pair of source graph Transitions that have an ObjectFlowState in between them. Here, it is interesting that the archetypes of the new vertices are no source model elements, but tuples of source model vertices. As already said, GReTL enforces no restriction on what can be used as archetypes. Choosing good archetypes is the main point in abstracting away special handling.

The next operation call creates one HasObject edge for each of those ObjectFlow archetype tuples. The tuple is also chosen as archetype for the new edges. The edges have to start at the images of the tuples, and those are the ObjectFlow vertices created in the previous operation. They should end at the images of the tuple's first Transitions target vertex. This is a source model ObjectFlowState, and for those ObjectNode vertices were created before (see end of section 2.1).

The ComesFrom and GoesTo edges will be instantiated uniformly for core and extension part in the *following shared part*.

Core Task. The small source model part around the ObjectFlowState for "Placed Order" (figure 4) transformed according to the core task is depicted in figure 6. The source and target structure are isomorphic, here. Transitions are transformed to ObjectFlows, and the ObjectFlowState v7 is represented by the ObjectNode v13.

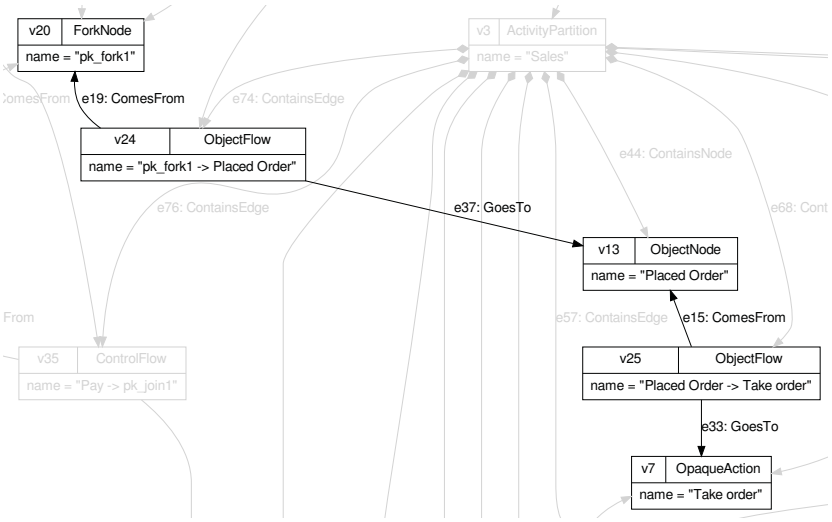


Figure 6: The source model part of figure 4 transformed according to the **core task**

In the core task, there is no need to create any HasObject edges, because the objects passed between actions are again modeled as ObjectNodes, but they are connected to the actions with one incoming and one outgoing ObjectFlow and usual ComesFrom and GoesTo edges, quite equivalent to the source model.

```
instantiateVertices(objectFlow,
    "from t : V{Transition}                                    "
  + "with not isEmpty(t -->{GoesTo, ComesFrom} & {ObjectFlowState}) "
  + "reportSet t, t end                                        ");
```

Here, one target ObjectFlow vertex is instantiated for each Transition either coming from or going to an ObjectFlowState vertex. Note that this are twice as many ObjectFlows as the extension task part creates. In order to have the same archetype structure as the extension part

and thus allowing the following shared operations to work for both of them, again a set of transition tuples is chosen as archetype set. Each tuple contains the Transition t twice.

These three invocations are the only variations needed to make the Activity1ToActivity2 transformation handle both the core and object flow extension part. As already said, the ComesFrom and GoesTo edges will be instantiated uniformly in the *following shared part*.

Following Shared Part. In this paragraph, the two invocations for creating the GoesTo and ComesFrom edge classes including instances thereof are presented, because they are also affected by the variation. But it has to be emphasized, that due to the selection of transition tuples as archetypes for ObjectFlow vertices, this variation is completely abstracted away.

The abstract edge class ActivityEdge and the ControlFlow edge class are created, and ActivityEdge is set as superclass of both ControlFlow and ObjectFlow.

```
1  VertexClass controlFlow = createVertexClass("ControlFlow",
2      "from t : V{Transition}                                    "
3    + "with isEmpty(t -->{ComesFrom, GoesTo} & {ObjectFlowState}) "
4    + "reportSet t, t end                                        ");
5  VertexClass activityEdge = createAbstractVertexClass("ActivityEdge");
6  addSubClasses(activityEdge, controlFlow, objectFlow);
```

For ControlFlow, Transition tuples containing the same transition twice are again used as archetypes, similar to the object flows in the core task part. This ensures that all archetypes in the domain of $img_{ActivityEdge}$ can be handled uniformly, no matter which task the transformation is currently handling.

With this setup, the creation of the ComesFrom and GoesTo edge classes and their instances in the target model is straight forward.

```
1  createEdgeClass("ComesFrom",
2      new IncidenceClassSpec(activityEdge),
3      new IncidenceClassSpec(vc("ActivityNode"), 1, 1),
4      "from t : keySet(img_ActivityEdge)                    "
5    + "reportSet t, t, theElement(t[0] -->{ComesFrom}) end");
6
7  createEdgeClass("GoesTo",
8      new IncidenceClassSpec(activityEdge),
9      new IncidenceClassSpec(vc("ActivityNode"), 1, 1),
10     "from t : keySet(img_ActivityEdge)                  "
11   + "reportSet t, t, theElement(t[1] -->{GoesTo}) end");
```

Both edge classes start at the ActivityEdge vertex class and lead to the ActivityNode edge class. At ActivityEdge, the default (0,*) multiplicity is used and (1,1) on the opposite side.

The semantic expressions iterate over the Transition tuples used as ActivityEdge archetypes, i. e. as archetypes of the concrete subclasses ControlFlow and ObjectFlow. Each ComesFrom / GoesTo edge has to start at the image of the tuple in $img_{ActivityEdge}$, which is either a ControlFlow or an ObjectFlow vertex. Each ComesFrom edge has to end at the image of the source of the tuple's first Transition, and each GoesTo has to end at the target of the tuple's second Transition. If the tuples contain the same Transition twice like it is the case for all ControlFlow archetypes and the archetypes of ObjectFlows in the core task, then the resulting structure is similar to the source model. But in the object flow extension task, a chain of two Transitions with on ObjectFlowState in between is transformed to exactly one ObjectFlow with connected ObjectNode.

3 Conclusion

In this paper, the GReTL transformation language was briefly introduced using the implementation of an UML 1.4 to UML 2.2 activity diagram transformation. This transformation creates the target metamodel on its own, instead of requiring an existing one. It is capable of performing the *Model Migration* case study's *core* as well as the *object flow extension task*. The variation between the two tasks could be narrowed down to the instantiation of two target metamodel types. By using tuples of source model transitions as archetypes for target model activity edges, it was possible to abstract away the differences, and all other operations are shared no matter which task is run.

It should be noted, that the transformation presented here is quite easy and doesn't show many of GReTL's benefits. In general, those show up when arbitrary complex, non-local structures have to be matched in the source graph. For example, in a reengineering project a GReTL transformation is used to extract state machines out of graphs conforming to a fine-granular Java schema. These graphs contain millions of vertices and edges and are syntaxgraph representations of the complete Java source code of the software system to be reengineered. To achieve its task, the transformation has to capture elements in method bodies in an arbitrary nesting depth, and also method call chains have to be followed transitively. Using GReQL's *regular path expressions*, which can also express transitive closures (see appendix A.1), the semantic expressions of that transformation are still very consise and specify the correlation between elements in a very natural, declarative manner.

A The TGraph Approach

In the TGraph approach approach, models are represented as TGraphs. Those are directed graphs with typed, attributed and ordered vertices and edges. Edges are first-class objects, which implies that they have an identity, they are typed, may have attributes, and they can always be traversed in both directions.

A visualization of a small part of the model migration case study's source TGraph is depicted in figure 7. There is an ActionState vertex with ID v4 and name "Request Service",

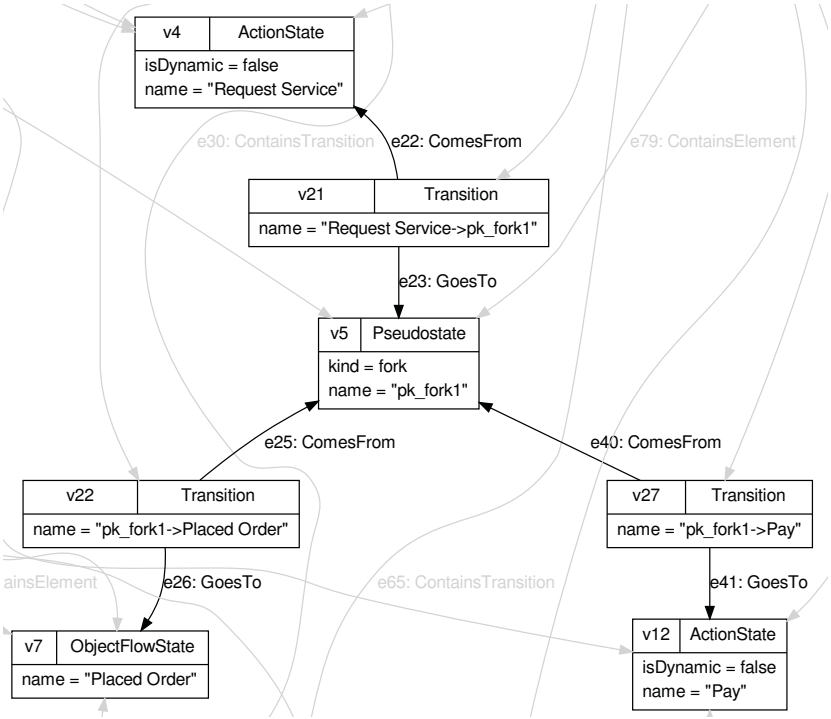


Figure 7: A visualization of a small part of the UML 1.4 source model

and a Pseudostate vertex v5 with kind set to "fork" and name set to "pk_fork1". The state v4 changes to the state v5 via the Transition vertex v21. That vertex is connected to v4 with the ComesFrom edge e22, and it is connected to the vertex v5 with the GoesTo edge e23. After the fork v5, control flow is split into two branches leading to the ObjectFlowState v7 via the Transition v22 and in parallel to the ActionState v12 via the Transition v27.

Each TGraph conforms to a *TGraph schema*, which is the metamodel of a class of TGraphs. Schemas are usually created using a profile of UML 2 class diagrams called *grUML* (*Graph UML*, [2]). Figure 8 shows the schema the UML 1.4 activity graph from figure 7 conforms to.

This schema was directly derived from the minimal UML 1.4 activity diagram Ecore model. The main difference is that all associations and compositions have an added name, which is the name of the edge type.

The schema specifies a graph class ADGraph. Such a graph may contain vertices of all the vertex types specified as UML classes. For example, an ADGraph may have vertices of

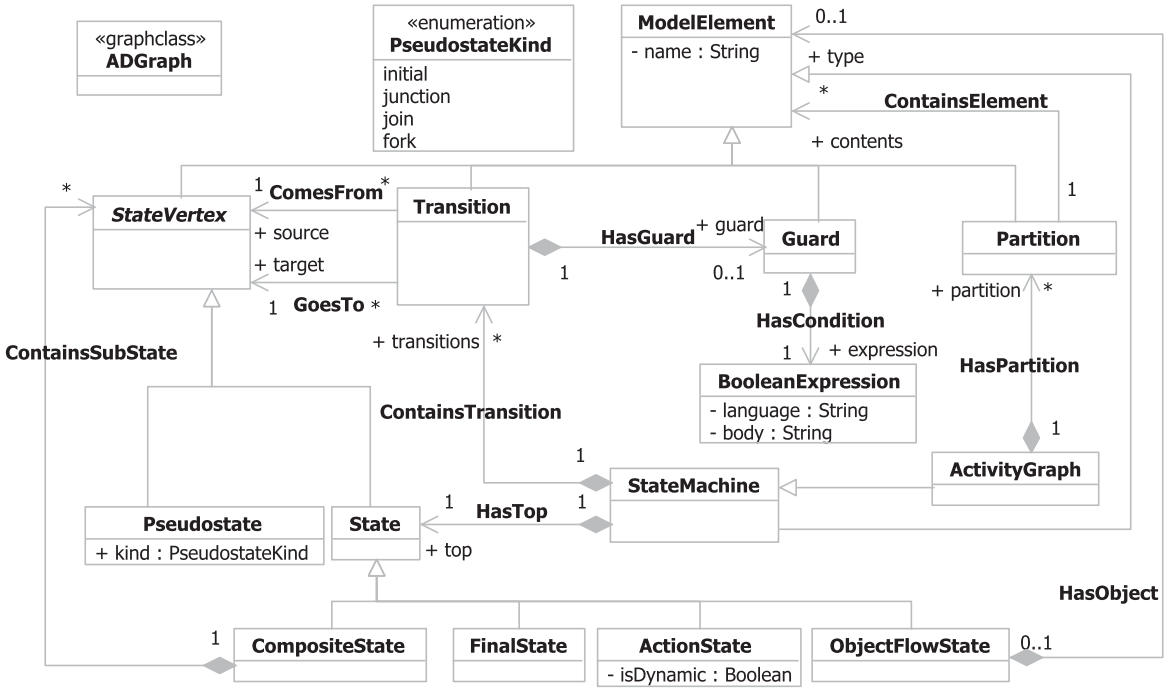


Figure 8: The UML 1.4 activity diagram source schema

type Transition, ActionState and Pseudostate, and it may contain edges of type ComesFrom and GoesTo, which run from Transition to StateVertex.

Each TGraph schema itself conforms to the *grUML metaschema*, which is the metameta-model of the TGraph approach. The metaschema is a valid schema describing itself. Its core is depicted in figure 9.

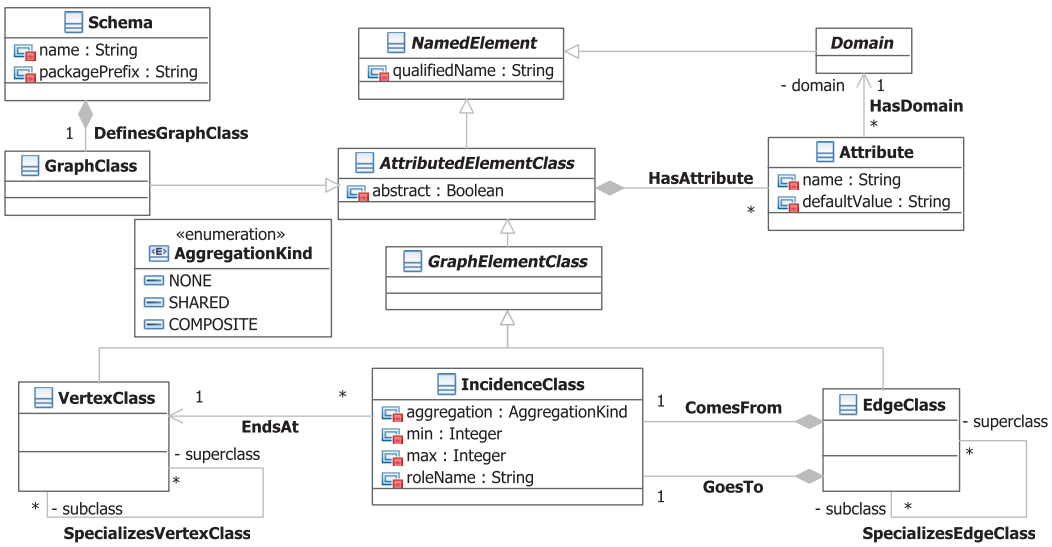


Figure 9: The grUML metaschema

Each Schema defines exactly one GraphClass, like the ADGraph in figure 8. Inside such a GraphClass, there are packages (hidden in the diagram) that contain GraphElementClasses. The two concrete forms are VertexClass and EdgeClass. Between both vertex as well as edge classes, there is support for specialization including multiple inheritance.

Association ends are modeled with IncidenceClasses. Each IncidenceClass belongs to exactly one VertexClass and to exactly one EdgeClass, and each EdgeClass has exactly one source and one target IncidenceClass, which hold this end's properties like multiplicities, role names, and the aggregation kind. There is also support for subsetting and redefinition of incidence classes [1], but the relevant associations are hidden in the figure.

The GraphClass and all Vertex- and EdgeClasses are AttributedElementClasses which may have Attributes. Each Attribute has a name, and the Domain specifies the type of its value. Supported are all primitive types known from Java, enumerations, and composite types like homogeneous sets, lists, tuples, maps and user-definable records.

The Java library *JGraLab*[5] provides a highly efficient API for accessing and manipulating TGraphs and TGraph schemas, code generation facilities, and many more components. Thus, it provides a seamless framework for model-based development.

A.1 Querying TGraphs With GReQL

The *Graph Repository Query Language* (*GReQL*, [4]) is a powerful model querying language for querying TGraphs. For GReTL, GReQL is what OCL [8] and its extensions is for most other model transformation languages.

One of the most commonly used language elements is the *from-with-report* (*FWR*) clause. The from part is used to declare *variables* and bind them to *domains*. In the with part, *constraints* can be imposed on the values of these variables. The report part is used to define the structure of the query result. A sample GReQL query is given in listing 1.

```
1 from elem : V{ModelElement}
2 with elem.name =~ ".*Service.*"
3 reportSet elem end
```

Listing 1: A simple GReQL query

Conceptually[6], the variable elem is bound to any vertex of type ModelElement or any subclass thereof one after the other. For each of those vertices, the constraint in the with part is checked. Here, it is checked if the value of the name attribute matches the regular expression[7] ".*Service.*". All vertices, for which this constraint evaluates to true are added to the result, which is a set (reportSet) in this case. When evaluated on the UML 1.4 source model from figure 7, it returns a set containing the vertices v4: ActionState, v20: Transition and v21: Transition. The vertices v4 and v21 are visible in that figure.

One of GReQL's unique and powerful features are *regular path expressions*, which can be used to formulate queries that utilize the structure of relationships between vertices. Therefore, symbols for edges (*path descriptions*) are introduced: --> and <-- for directed edges,

[5] http://jgralab.uni-koblenz.de

[6] Query optimization is not considered here.

[7] Java regular expressions are used here, see java.util.regex.

<–> if the direction is not considered, and <>–– and ––<> for edges with aggregation or composition semantics. Additionally, an edge type or role name written in curly braces the edge symbol to restrict the search to certain edge types. These symbols can be combined using regular operators: sequence, iteration (*, +, and ^n), alternative (|), and transposition (^T).

The query in listing 2 uses such a regular path expression to calculate all successor states of the ActionState v4 from figure 7. Using the let expression, the variable requestService is

```
1 let requestService := getVertex(4) in
2   from succState : requestService (<––{ComesFrom} ––>{GoesTo})+ & {State}
3   reportSet succState, succState.name end
```

Listing 2: Retrieving all successor states of ActionState v4

bound to the vertex with ID 4. Comparing with figure 7, this is the ActionState "Request Service". Then, the variable succState is bound to all successor states one after the other. Those are calculated using a *forward vertex set*. The anchor is the vertex bound to requestService. From that vertex, one or many (+) sequences of a ComesFrom followed by a GoesTo edge may be traversed. The *goal restriction* & State restricts the result vertices to the type State or any subtype thereof. For each successor state, the state itself and the value of its name attribute is reported.

When evaluated on the source graph from figure 7, the query retrieves the following result:

```
1 (v7: ObjectFlowState, Placed Order)      | (v17: ActionState, Restock)
2 (v12: ActionState, Pay)                  | (v14: ActionState, Collect order)
3 (v6: ActionState, Take order)            | (v8: ActionState, Fill Order)
4 (v9: ObjectFlowState, Entered Order)     | (v19: FinalState, Finished)
5 (v13: ActionState, Deliver order)        | (v11: ObjectFlowState, Filled Order)
6 (v15: ObjectFlowState, Delivered Order)  |
```

Comparing with the source graph of figure 7, starting form vertex v4 the first vertex reachable via traversing a sequence of incoming ComesFrom and then outgoing GoesTo edges leads to the Pseudostate v5. But according to the source schema in figure 8, a Pseudostate is not a State, and so v5 is not in the result set. But with one more ComesFrom/GoesTo edge sequence traversal, the ObjectFlowState v7 and the ActionState v12 are reached. Both ObjectFlowState an ActionState are subtypes of State, and so both vertices are contained in the result set. The following result set elements can be reached with further iterations, but they are not visible anymore in figure 7.

B References

[1] Daniel Bildhauer. On the Relationships Between Subsetting, Redefinition and Association Specialization. In *Communications of the Ninth International Baltic Conference on Databases and Information Systems*. To Appear, 2010.

[2] Daniel Bildhauer, Tassilo Horn, Volker Riediger, Hannes Schwarz, and Sascha Strauß. *grUML - A UML based modeling language for TGraphs*, 2010. unpublished.

[3] J. Ebert, V. Riediger, and A. Winter. Graph Technology in Reverse Engineering, The TGraph Approach. In R. Gimnich, U. Kaiser, J. Quante, and A. Winter, editors, *10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics*, pages 67–81. GI, 2008.

[4] Jürgen Ebert and Daniel Bildhauer. Reverse Engineering Using Graph Queries. In Andy Schürr, Claus Lewerentz, Gregor Engels, Wilhelm Schäfer, and Bernhard Westfechtel, editors, *Graph Transformations and Model Driven Engineering*, LNCS 5765. Springer, 2010. to appear.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[6] Tassilo Horn and Jürgen Ebert. The GReTL Transformation Language. Technical report, University Koblenz-Landau, Institute for Software Technology, 2010. unpublished, draft at `http://www.uni-koblenz.de/~horn/gretl.pdf`.

[7] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.

[8] OMG. Object Constraint Language Version 2.0, 2006.

C The Transformation Source Code

The following listing shows the complete source code of the Activity1ToActivity2 transformation. When only counting the transformation code, i.e. no comments, empty lines and the main() method, the whole transformation takes about 120 lines of code to create the target UML 2.2 activity schema and migrate arbitrary source models conforming to the UML 1.4 activity schema to the new metamodel.

```
1  public class Activity1ToActivity2 extends Transformation {
2      public enum Task { CORE, OBJECT_FLOW_EXTENSION };
3      private Task task;
4      public void setTask(Task task) { this.task = task; }
5      public Activity1ToActivity2(Context c) { super(c); }
6
7      @Override protected void transform() {
8          // This array contains pairs {NewType, OldType}.
9          //
10         // Schema Level: For each OldType vertex class, a new vertex class with qualified
11         // name NewType is created in the target schema.
12         //
13         // Instance Level: For each source graph OldType vertex, a new NewType vertex is
14         // created in the target graph.
15         for (String[] s : new String[][] { { "Activity", "ActivityGraph" },
16                 { "ActivityPartition", "Partition" }, { "ActivityFinalNode", "FinalState" },
17                 { "OpaqueAction", "ActionState" }, { "ObjectNode", "ObjectFlowState" },
18                 { "OpaqueExpression", "Guard" } }) {
19             createVertexClass(s[0], "V{" + s[1] + "}");
20         }
21
22         // Initial-, Fork-, Join-, DecisionNodes are all Pseudostates in UML1.4, only
23         // distinguishable by their kind attribute. So this array contains pairs {NewType,
24         // kindAttrValue}.
25         //
26         // Schema Level: For each array element, create a vertex class with qualified name
27         // NewType in the target metamodel.
28         //
29         // Instance Level: For each source model Pseudostate with kind = kindAttrValue,
30         // create one target model vertex of type NewType.
31         for (String[] s : new String[][] { { "InitialNode", "initial" },
32                 { "ForkNode", "fork" }, { "JoinNode", "join" },
33                 { "DecisionNode", "junction" } }) {
34             createVertexClass(s[0],
35                     "from ps : V{Pseudostate}              "
36                   + "with ps.kind = \"" + s[1] + "\"       "
37                   + "reportSet ps end                      ");
38         }
39
40         // Schema Level: Create an abstract vertex class ActivityNode in the target schema.
41         //
42         // Instance Level: Abstract classes don't have instances, so this operation doesn't
43         // affect the instance level.
44         VertexClass activityNode = createAbstractVertexClass("ActivityNode");
45
46         // Schema Level: Make ActivityNode the superclass of all the following 6 vertex
47         // classes that were already created by the previous operation calls. The method
48         // vc(String) simply retrieves the target metamodel vertex class with the given
49         // qualified name.
```

```
50          //
51          // Instance Level: No effect.
52          addSubClasses(activityNode, vc("OpaqueAction"), vc("InitialNode"),
53                  vc("ActivityFinalNode"), vc("DecisionNode"), vc("JoinNode"),
54                  vc("ForkNode"), vc("ObjectNode"));
55
56          // Schema Level: Create the vertex class ObjectFlow in the target schema.
57          //
58          // Instance Level: Different in CORE and OBJECT_FLOW_EXTENSION task, so the instance
59          // creation handling is done below in the switch, and here no instances are created
60          // at all.
61          VertexClass objectFlow = createVertexClass("ObjectFlow");
62
63          // Schema Level: Create a new edge class HasObject with composition semantics from
64          // ObjectFlow to ObjectNode in the target schema.  ObjectFlow is the whole and
65          // ObjectNode is on the part side.
66          //
67          // ObjectFlow (0,1) <>-HasObject-> (0,1) ObjectNode
68          //
69          // Instance Level: Instances are only needed in the extension task, but not in the
70          // core task.
71          EdgeClass hasObject = createEdgeClass("HasObject",
72                  new IncidenceClassSpec(objectFlow, 0, 1),
73                  new IncidenceClassSpec(vc("ObjectNode"), 0, 1, AggregationKind.COMPOSITE));
74
75          // The following operation calls vary between CORE and OBJECT_FLOW_EXTENSION task.
76          switch (task) {
77          case CORE:
78              // Schema Level: already done above, so nothing to to here.
79              //
80              // Instance Level: For each Transition vertex t, which goes to or comes from an
81              // ObjectFlowState, create one ObjectFlow vertex in the target graph. As
82              // archetype, we use a tuple where both components are the transition t, because
83              // this makes the structure uniform to the OBJECT_FLOW_EXTENSION task.
84              instantiateVertices(objectFlow,
85                      "from t : V{Transition}                                          "
86                    + "with not isEmpty(t -->{GoesTo, ComesFrom} & {ObjectFlowState}) "
87                    + "reportSet t, t end                                            ");
88              break;
89          case OBJECT_FLOW_EXTENSION:
90              // Schema Level: already done above, so nothing to to here.
91              //
92              // Instance Level: For each pair of Transition vertices (t1, t2), which build up
93              // a source model structure like
94              //
95              // t1 -->{GoesTo} & {ObjectFlowState} <--{ComesFrom} t2
96              //
97              // create one ObjectFlow vertex in the target graph. Use the tuple (t1, t2) as
98              // archetype for the new object flow.
99              instantiateVertices(objectFlow,
100                     "from t1 : V{Transition}, t2 : V{Transition}              "
101                   + "with t1 -->{GoesTo} & {ObjectFlowState} <--{ComesFrom} t2 "
102                   + "reportSet t1, t2 end                                      ");
103
104             // Schema Level: already done above, so nothing to to here.
105             //
106             // Instance Level: For each Transition tuple t in the domain of img_ObjectFlow
```

```
107             // create one HasObject edge. It starts at the image of the tuple t which is an
108             // ObjectFlow, and it ends at the image of the tuple's first Transition's
109             // GoesTo-target, which is some ObjectFlowState for which we've already created
110             // an ObjectNode.
111             instantiateEdges(hasObject,
112                     "from t : keySet(img_ObjectFlow)                    "
113                   + "reportSet t, t, theElement(t[0] -->{GoesTo}) end ");
114             break;
115         default:
116             throw new GReTLException(context, "Unknown task '" + task + "'!");
117         }
118         // That was the whole task-specific part. Due to the uniform selection of archetypes,
119         // the rest can be handled uniformly.
120
121         // Schema Level: Create a vertex class ControlFlow in the target schema.
122         //
123         // Instance Level: For each source model Transition which doesn't start or end at an
124         // ObjectFlowState create one ControlFlow vertex in the target graph. Nevertheless,
125         // we use transition tuples as archetypes, so that img_ControlFlow's domain is
126         // structurally equal to the domain of img_ObjectFlow, that is (Transition x
127         // Transition).
128         VertexClass controlFlow = createVertexClass("ControlFlow",
129                 "from t : V{Transition}                                    "
130               + "with isEmpty(t -->{ComesFrom, GoesTo} & {ObjectFlowState}) "
131               + "reportSet t, t end                                        ");
132
133         // Schema Level: Create the abstract vertex class ActivityEdge.
134         //
135         // Instance Level: No effect.
136         VertexClass activityEdge = createAbstractVertexClass("ActivityEdge");
137
138         // Schema Level: Make ControlFlow and ObjectFlow specializations of ActivityEdge.
139         //
140         // Instance Level: No effect.
141         addSubClasses(activityEdge, controlFlow, objectFlow);
142
143         // Schema Level: Creates the composition edge class HasGuard.
144         //
145         // ActivityEdge (0,1) <>-HasGuard-> (0,1) OpaqueExpression
146         //
147         // Instance Level: For each HasGuard edge in the source model create a HasGuard edge
148         // in the target model. The start vertex is the image of a tuple
149         // (connectedTransition, connectedTransition), and that's exactly the archetype we
150         // chose for ActivityEdges. The end vertex is the image of the edge's end vertex,
151         // which is a Guard. Above, we created one OpaqueExpression for each Guard.
152         createEdgeClass("HasGuard",
153              new IncidenceClassSpec(activityEdge, 0, 1),
154              new IncidenceClassSpec(vc("OpaqueExpression"), 0, 1, AggregationKind.COMPOSITE),
155             "from e : E{HasGuard}                                          "
156          + "reportSet e, tup(startVertex(e), startVertex(e)), endVertex(e) end");
157
158         // Schema Level: Create the body attribute of type String for the vertex class
159         // OpaqueExpression.
160         //
161         // Instance Level: Simply set the value according the Guard g's source model
162         // condition body, which is some BooleanExpression.
163         createAttribute(new AttributeSpec(vc("OpaqueExpression"), "body", getStringDomain()),
```

```
                "from g : V{Guard}                                    "
              + "reportMap g, theElement(g-->{HasCondition}).body end");

        // Schema Level: Create the language attribute of type String for the vertex class
        // OpaqueExpression. Use the string "natural" as default value. (In the input model,
        // the language attribute isn't set for any element.)
        //
        // Instance Level: Simply set the value according the Guard g's source model
        // condition language unless that is not set. In that case, stick with the default
        // value.
        //
        // In our source model, there's no value set for any BooleanExpression's language
        // attribute, so here we also handle a case which doesn't really occur...
        createAttribute(new AttributeSpec(vc("OpaqueExpression"), "language",
                                          getStringDomain(), "\"natural\""),
                "from g : V{Guard}                                    "
              + "with oldValue <> null                                "
              + "reportMap g, oldValue end                            "
              + "where oldValue := theElement(g-->{HasCondition}).language ");

        // Schema Level: Create an edge class ComesFrom.
        //
        // ActivityEdge (0,*) --ComesFrom-> (1,1) ActivityNode
        //
        // Instance Level: For each tuple t with contents (t0:Transition, t1:Transition) in
        // the domain of img_ActivityEdge create one ComesFrom edge. This edge starts at the
        // image of the tuple (some ActivityEdge), and it ends at the image of the Transition
        // t0's neighbor vertex connected with a ComesFrom edge.
        createEdgeClass("ComesFrom",
                  new IncidenceClassSpec(activityEdge),
                  new IncidenceClassSpec(vc("ActivityNode"), 1, 1),
                "from t : keySet(img_ActivityEdge)                    "
              + "reportSet t, t, theElement(t[0] -->{ComesFrom}) end");

        // Schema Level: Create an edge class GoesTo.
        //
        // ActivityEdge (0,*) --GoesTo-> (1,1) ActivityNode
        //
        // Instance Level: For each tuple t with contents (t0:Transition, t1:Transition) in
        // the domain of img_ActivityEdge create one GoesTo edge. This edge starts at the
        // image of the tuple (some ActivityEdge), and it ends at the image of the Transition
        // t1's neighbor vertex connected with a GoesTo edge.
        createEdgeClass("GoesTo",
                  new IncidenceClassSpec(activityEdge),
                  new IncidenceClassSpec(vc("ActivityNode"), 1, 1),
                "from t : keySet(img_ActivityEdge)                    "
              + "reportSet t, t, theElement(t[1] -->{GoesTo}) end");

        // Schema Level: Create a composition edge class ContainsNode.
        //
        // ActivityPartition (0,1) <>-ContainsNode-- (0,*) ActivityNode
        //
        // Instance Level: For each source model ContainsElement edge, which is targeting
        // some ModelElement which was transformed to some ActivityNode, create a
        // ContainsNode edge. It starts and ends at the images of the edge e's start and end
        // vertices.
        createEdgeClass("ContainsNode",
```

```
221                 new IncidenceClassSpec(vc("ActivityPartition"), 0, 1),
222                 new IncidenceClassSpec(vc("ActivityNode"), AggregationKind.COMPOSITE),
223               "from e : E{ContainsElement}                                    "
224             + "with containsKey(img_ActivityNode, endVertex(e))               "
225             + "reportSet endVertex(e), startVertex(e), endVertex(e) end");
226
227         // Schema Level: Create a composition edge class ContainsEdge.
228         //
229         // ActivityPartition (0,1) <>-ContainsEdge-- (0,*) ActivityEdge
230         //
231         // Instance Level: The source model has no ContainsElement edges from Partitions to
232         // Transitions, so we add some heuristics here. Each ActivityEdge will be contained
233         // in all ActivityPartitions of its source and target ActivityNode.
234         createEdgeClass("ContainsEdge",
235                 new IncidenceClassSpec(vc("ActivityPartition"), 0, 1),
236                 new IncidenceClassSpec(activityEdge, AggregationKind.COMPOSITE),
237               "from t : keySet(img_ActivityEdge),                                "
238             + "     p : flatten(from s : union(t[0]-->{ComesFrom}, t[1]-->{GoesTo}) "
239             + "                 reportSet s <--{ContainsElement} end)           "
240             + "reportSet tup(t, p), p, t end                                    ");
241
242         // Schema Level: Create a composition edge class ActivityContainsGroup.
243         //
244         // Activity (1,1) <>-ActivityContainsGroup-- (0,*) ActivityPartition
245         //
246         // Instance Level: For each source model HasPartition edge, create an
247         // ActivityContainsGroup edge starting at the image of this edges start vertex (an
248         // ActivityGraph) to the image of this edge's end vertex (a Partition).
249         createEdgeClass("ActivityContainsGroup",
250                 new IncidenceClassSpec(vc("Activity")),
251                 new IncidenceClassSpec(vc("ActivityPartition"), AggregationKind.COMPOSITE),
252               "from e : E{HasPartition}                                       "
253             + "reportSet e, startVertex(e), endVertex(e) end ");
254
255         // Schema Level: Create a composition edge class ActivityContainsNode.
256         //
257         // Activity (0,1) <>-ActivityContainsNode-- (0,*) ActivityNode
258         //
259         // Instance Level: For each ActivityNode archetype (some StateVertex) which is
260         // connected to no ContainsElement edge, create one ActivityContainsNode from the
261         // image of the nearest ActivityGraph containing this archetype to its own image.
262         //
263         // The constraint ensures that no ActivityContainsNode edges are created for
264         // ActivityNodes which are already contained in an ActivityPartition which in turn is
265         // contained in an Activity.
266         createEdgeClass("ActivityContainsNode",
267                 new IncidenceClassSpec(vc("Activity"), 0, 1),
268                 new IncidenceClassSpec(activityNode, AggregationKind.COMPOSITE),
269               "from a : keySet(img_ActivityNode)                              "
270             + "with degree{ContainsElement}(a) = 0                            "
271             + "reportSet a, theElement(a --<>* & {ActivityGraph}), a end");
272
273         // Schema Level: Create a composition edge class ActivityContainsEdge.
274         //
275         // Activity (0,1) <>-ActivityContainsEdge-- (0,*) ActivityEdge
276         //
277         // Instance Level: For each ActivityEdge archetype (those are (Transition,
```

```
278         // Transition) tuples), for which no ContainsEdge has already been created, create
279         // one ActivityContainsEdge from the image of the nearest ActivityGraph containing
280         // the transition t[0] to the image of the tuple itself, which is an ActivityEdge.
281         //
282         // Again, the condition in the with-clause ensures that no ActivityContainsEdge edges
283         // are created for ActivityEdes which are already contained in an ActivityPartition
284         // which in turn is contained in an Activity.
285         createEdgeClass("ActivityContainsEdge",
286                   new IncidenceClassSpec(vc("Activity"), 0, 1),
287                   new IncidenceClassSpec(activityEdge, AggregationKind.COMPOSITE),
288                 "from t : keySet(img_ActivityEdge)                              "
289               + "with isEmpty(from x : keySet(img_ContainsEdge)                 "
290               + "             with x[0] = t                                     "
291               + "             reportSet x end)                                  "
292               + "reportSet t, theElement(t[0] --<>* & {ActivityGraph}), t end");
293
294         // Schema Level: Create the abstract vertex class ModelElement.
295         //
296         // Instance Level: No effect.
297         VertexClass modelElement = createAbstractVertexClass("ModelElement");
298
299         // Schema Level: Make ModelElement the superclass of ActivityEdge, Activity,
300         // ActivityPartition, ActivityNode and ObjectNode.
301         //
302         // Instance Level: No effect.
303         addSubClasses(modelElement, activityEdge, vc("Activity"),
304                 vc("ActivityPartition"), vc("ActivityNode"));
305
306         // Schema Level: Create a name attribute of type String at the ModelElement vertex
307         // class.
308         //
309         // Instance Level: For each target ModelElement excluding ActivityEdges, set the
310         // value of the attribute according to the value of the archetype.
311         Attribute name = createAttribute(
312                   new AttributeSpec(modelElement, "name", getStringDomain()),
313                 "from me : difference(keySet(img_ModelElement), "
314               + "                     keySet(img_ActivityEdge)) "
315               + "reportMap me, me.name end                        ");
316
317         // Schema Level: No effect.
318         //
319         // Instance Level: For each ActivityEdge archetype, set the name of the image
320         // according their predecessor/successor-ActivityNode. So if an activity edge starts
321         // at an "A" action and goes to a "B" action, the ActivityEdge's name will be "A ->
322         // B". The GReQL operator ++ is used to concatenate two strings.
323         instantiateAttributeValues(name,
324                   "from t : keySet(img_ActivityEdge)                    "
325                 + "reportMap t, theElement(t[0]-->{ComesFrom}).name     "
326                 + "             ++ \" -> \"                             "
327                 + "             ++ theElement(t[1]-->{GoesTo}).name end");
328     }
329
330     public static void main(String[] args) throws GraphIOException {
331         // Create a new Context object holding the state of the transformation. The name of
332         // the target schema to be created by the transformation is
333         // de.uni_koblenz.uml2.ActivitySchema, and it will define the graph class
334         // ActivityGraph.
```

```
335         Context c = new Context("de.uni_koblenz.uml2.ActivitySchema", "ActivityGraph");
336
337         // Set the source model for the transformation. The source graph is created
338         // programmatically by the CreateUML1ActivityGraph class in the util package. You
339         // might want to have a look.
340         c.setSourceGraph(CreateUML1ActivityGraph.getUml1ADGraph());
341
342         // Instantiate our transformation.
343         Activity1ToActivity2 acti1ToActi2 = new Activity1ToActivity2(c);
344
345         // Run the transformation once with CORE and the second time with
346         // OBJECT_FLOW_EXTENSION task. It will create the UML2 target metamodel/schema and in
347         // the same time migrate the UML1.4 model to a target model conforming to the just
348         // created UML2.2 schema.
349         for (Task task : new Task[] { Task.CORE, Task.OBJECT_FLOW_EXTENSION }) {
350             // Reset the context (that won't forget the target schema, so that the
351             // transformation will reuse the schema created in the first run).
352             c.reset(false);
353             // Set the current task.
354             acti1ToActi2.setTask(task);
355             // Execute the transformation.
356             acti1ToActi2.execute();
357             // Retrieve the target model.
358             Graph targetGraph = c.getTargetGraph();
359             // Save the target model in the native TG format.
360             GraphIO.saveGraphToFile("uml2model_" + task.toString().toLowerCase() + ".tg",
361                     targetGraph, new ProgressFunctionImpl());
362             // Also save a visualization of the target model in the GraphViz DOT format. You
363             // can view that using GraphViz' dotty program.
364             Tg2Dot.printGraphAsDot(targetGraph, false, "uml2model_" + task.toString()
365               .toLowerCase() + ".dot", targetGraph.getSchema()
366               .getAttributedElementClass("ComesFrom").getM1Class());
367         }
368     }
369 }
```

Migrating Activity Diagrams with Epsilon Flock

Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science, University of York, UK.
[louis,dkolovos,paige,fiona]@cs.york.ac.uk

Abstract. The Transformation Tools Contest 2010 workshop invites solutions to a model migration case in which UML activity diagrams are to be migrated from UML 1.4 to UML 2.2. This paper presents a solution to this case, which uses Epsilon Flock, a model transformation language tailored for model migration.

1 Introduction

In [4], we propose a case for exploring and comparing the ways in which model migration can be specified, using an example from the evolution of UML. The way in which activity diagrams are modelled in the UML has changed significantly between versions 1.4 and 2.2 of the specification. In this paper, we briefly introduce Epsilon Flock [5], a model transformation language tailored for model migration, and present our solutions to the case described in [4].

2 Epsilon Flock

Epsilon Flock [5] (subsequently referred to as Flock) is a model transformation language tailored for model migration. In particular, Flock automatically copies from original to migrated model all model elements that have not been affected by metamodel evolution. The user need only specify the migration strategy for those model elements that do not conform to the evolved metamodel. Flock is built atop Epsilon[1] [1], an extensible platform providing inter-operable programming languages for model-driven development. Epsilon, and hence Flock, can be used with a range of modelling technologies, such as EMF [6], MDR, Z and XML.

3 Core Task

The core task described by [4] requires submissions to migrate an activity diagram that conforms to a (minimal) UML 1.4 metamodel to an equivalent activity diagram conforming to the UML 2.2 metamodel provided by the Eclipse UML 2 tools project. We approached this problem in an iterative and incremental manner, using the following process:

[1] `http://www.eclipse.org/gmt/epsilon`

1. Change the Flock migration strategy.
2. Execute Flock on the original model, producing a migrated model.
3. Compare the migrated model with the reference model provided in [4].
4. Repeat until the migrated and reference models were the same.

The remainder of this section presents our Flock solution in an incremental manner. The code listings in this section show only those rules relevant to the iteration being discussed. The complete migration strategy is shown in Appendix B.

3.1 Actions, Transitions and Final States

We started by executing Flock on the original model with an empty migration strategy. The resulting model contained `Pseudostatess` and `Transitions`, but none of the `ActionStates` from the original model. In UML 2.2 activities, `OpaqueActions` replace `ActionStates`. Listing 1.1 shows the Flock code for changing `ActionStates` to corresponding `OpaqueActions`.

```
1  migrate ActionState to OpaqueAction
```

Listing 1.1. Migrating Actions

Listing 1.1 contains a single *migration rule*, the fundamental building block of a Flock migration strategy. Each rule specifies an original type (`ActionState`) and an optional evolved type (`OpaqueAction`). Rules can also specify an optional guard (a boolean expression preceded with the `when` keyword). Flock will execute the rule in Listing 1.1 once for each `ActionState` in the original model, producing one `OpaqueAction` in the migrated model.

We added similar rules to the Flock migration strategy to migrate instances of `FinalState` to instances of `ActivityFinalNode` and to migrate instances of `Transition` to `ControlFlow`, as shown in Listing 1.2.

```
1  migrate FinalState to ActivityFinalNode
2  migrate Transition to ControlFlow
```

Listing 1.2. Migrating FinalStates and Transitions

3.2 Pseudostates

Next, we codified migration for `Pseudostatess`, which are no longer used in UML 2.2 activities. Instead, UML 2.2 activities use specialised `Nodes`, such as `InitialNode`. Listing 1.3 shows the Flock code for changing `Pseudostates` to corresponding `Nodes`.

```
1  migrate Pseudostate to InitialNode when: original.kind = Original!
       PseudostateKind#initial
2  migrate Pseudostate to DecisionNode when: original.kind = Original!
       PseudostateKind#junction
3  migrate Pseudostate to ForkNode when: original.kind = Original!
       PseudostateKind#fork
4  migrate Pseudostate to JoinNode when: original.kind = Original!
       PseudostateKind#join
```

Listing 1.3. Migrating Pseudostates

Listing 1.3 contains four migration rules, which migrate `Pseudostates` to some subtype of `Node` (`InitialNode`, `DecisionNode`, etc) based on the value of the original model element's kind feature. For example, the first migration rule in Listing 1.3 states that a `Pseudostate` with `kind` equal to to the `initial` element of the `PseudostateKind` enumeration will be migrated by producing an `InitialNode`.

For each element of the original model, Flock will execute at most one *applicable* migration rule. For example, the rule show on line 1 of Listing 1.3 is applicable to every original model element that is an instance of `Pseudostate` and whose kind attribute is equal to the `initial` element of the `PseudostateKind` enumeration. The way in which Flock selects applicable rules is discussed more thoroughly in [5].

3.3 Activities

In UML 2.2, `Activitys` no longer inherit from state machines. As such, some of the features defined by `Activity` have been renamed. Specifically, `transitions` has become `edges` and `paritions` has become `group`. Furthermore, the states (or nodes in UML 2.2 parlance) of an `Activity` are now contained in a feature called `nodes`, rather than in the `subvertex` feature of a composite state accessed via the `top` feature of `Activity`. The Flock migration rule shown in Listing 1.4 captures these changes.

```
1  migrate ActivityGraph to Activity {
2    migrated.edge = original.transitions.equivalent();
3    migrated.group = original.partition.equivalent();
4    migrated.node = original.top.subvertex.equivalent();
5  }
```

Listing 1.4. Migrating ActivityGraphs

Listing 1.4 contains one migration rule, which produces an instance of `Activity` from an instance of `ActivityGraph`. The rule specifies no guard, and so is applicable to all instances of `ActivityGraph`. The body of the rule (lines 2-4) is executed for each `ActivityGraph` in the original model. The Epsilon Object Language (EOL) [2] is used to specify the body of Flock migration rules. EOL is a reworking and extension of OCL that includes the ability to update models, conditional and loop statements, statement sequencing, and access to standard I/O streams.

Flock makes available two variables for use in the body of rules: `original` and `migrated` which can be used to access the original and migrated model elements. Here, the body of the rule copies values from the original to the migrated model element. For instance, line 2 copies the contents of the transitions feature to the edge feature. Because `original.transitions` will return a collection of original model elements, the built-in `equivalent` operation is used to find the equivalent migrated model elements. The `equivalent` operation invokes other migration rules where necessary, caching results where possible to improve performance.

We added a similar rule for migrating Guards. In UML 1.4, the the guard feature of Transition references a Guard, which in turn references an Expression via its expression feature. In UML 2.2, the guard feature of Transition references an OpaqueExpression directly. Listing 1.5 captures this in Flock.

```
1  migrate Guard to OpaqueExpression {
2    migrated.body.add(original.expression.body);
3  }
```

Listing 1.5. Migrating Guards

3.4 Partitions

In UML 1.4 activity diagrams, Partition specifies a single containment reference for it contents. In UML 2.2 activity diagrams, partitions (termed ActivityPartitions) specify two containment features for their contents, edges and nodes. Listing 1.6 captures this change in Flock. The body of the rule shown in Listing 1.6 uses the collect operation to segregate the contents feature of the original model element into two parts.

```
1  migrate Partition to ActivityPartition {
2    migrated.edges = original.contents.collect(e:Transition | e.equivalent());
3    migrated.nodes = original.contents.collect(n:StateVertex | n.equivalent());
4  }
```

Listing 1.6. Migrating Partitions

3.5 ObjectFlows

Finally, we codified migration for object flows. In UML 1.4 activity diagrams, object flows (used to model activities that effect a change to an object) are specified using ObjectFlowState, a subtype of StateVertex. In UML 2.2 activity diagrams, object flows are modelled using a subtype of ObjectNode. In UML 2.2 flows that connect to and from ObjectNodes must be represented with ObjectFlows rather than ControlFlows.

Listing 1.7 shows the Flock source code used to migrate Transitons to ObjectFlows. The rule applies for Transitions whose source or target StateVertex is of type ObjectFlowState.

```
1  migrate ObjectFlowState to ActivityParameterNode
2
3  migrate Transition to ObjectFlow when: original.source.isTypeOf(
       ObjectFlowState) or original.target.isTypeOf(ObjectFlowState)
```

Listing 1.7. Migrating ObjectFlows

Extensions to the core task are discussed in Appendix A. The complete Flock code listing for the core task is shown in Appendix B.

4 Discussion

Application of Flock to the UML activity diagram example has highlighted strengths and weakness of the approach. Flock provides a *compact and familiar syntax* for expressing migration, For the body of migration rules, Flock re-uses the Epsilon Object Language (EOL) [2], a re-working and extension of OCL. Applying Flock to this example has highlighted *duplication caused by the current syntax* (e.g. see Listing 1.10), which could be simplified.

Because Flock automatically copies model elements that are unaffected by evolution, Flock supports *iterative and incremental development* of migration strategies. For analysing metamodel changes, the *Flock development tools could be enhanced*, for example with a metamodel differencing view.

Flock works with an *extensible range of modelling technologies* in a transparent manner. In Section A.2, we demonstrated this claim by re-using a migration strategy built for EMF models for migrating an MDR model.

Acknowledgement. The work in this paper was supported by the European Commission via the MADES project under the "Information Society Technologies" Seventh Framework Programme (2009-2012).

References

1. D.S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, United Kingdom, 2009.
2. D.S. Kolovos, R.F. Paige, and F.A.C Polack. The Epsilon Object Language (EOL). In *Proc. ECMDA-FA*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.
3. OMG. Unified Modelling Language 1.4 Specification [online]. [Accessed 5 March 2010] Available at: `http://www.omg.org/spec/UML/1.4/`, 2001.
4. L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model migration case for TTC 2010. In *Proc. TTC Workshop [accepted and to appear]*, 2010.
5. L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model migration with Epsilon Flock. In *Proc. ICMT [accepted and to appear]*, 2010.
6. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

A Extensions

In the case description [4], three extensions are discussed. Solutions for two of the extensions are described in this appendix.

A.1 Alternative ObjectFlowState Migration Semantics

The first extension described in [4] requires submissions to consider an alternative migration semantics for ObjectFlowState, in which a single ObjectFlow replaces each ObjectFlowState and any connected Transitions.

Listing 1.8 shows the Flock source code used to migrate ObjectFlowStates (and connecting Transitions) to a single ObjectFlow. This rule replaces the two rules defined in Listing 1.7. In the body of the rule, the `source` of the `Transition` is copied directly to the `source` of the `ObjectFlow`. The `target` of the `ObjectFlow` is set to the target of the first outgoing `Transition` from the `ObjectFlowState`.

```
1  migrate Transition to ObjectFlow when: original.target.isTypeOf(
       ObjectFlowState) {
2    migrated.source = original.source.equivalent();
3    migrated.target = original.target.outgoing.first.target.equivalent();
4  }
```

Listing 1.8. Migrating ObjectFlowStates to a single ObjectFlow

Because `ObjectFlowStates` are represented as edges rather than nodes, the partition migration rule was changed such that `ObjectFlowStates` were not copied to the `nodes` feature of `Partitions`, as shown on line 12 of Listing 1.9 which replaces the migration rule shown in Listing 1.6.

```
1  migrate Partition to ActivityPartition {
2    migrated.edges = original.contents.collect(e:Transition | e.equivalent());
3    migrated.nodes = original.contents.reject(ofs:ObjectFlowState | true).
         collect(n:Original!StateVertex | n.equivalent());
4  }
```

Listing 1.9. Migrating Partitions without ObjectFlowStates

A.2 XMI

The second extension described in [4] requires submissions to migrate an activity graph conforming to the UML 1.4 metamodel provided by the OMG [3] to an equivalent activity diagram conforming to the UML 2.2 metamodel provided by the Eclipse UML 2 tools project. Crucially for this task, UML 1.4 models are specified in XMI 1.2, while UML 2.2 models are specified in XMI 2.1.

Flock is built atop Epsilon, which includes a model connectivity layer (EMC). EMC provides a common interface for accessing and persisting models. Currently, EMC provides drivers for several modelling frameworks, permitting management of models defined with EMF, the Metadata Repository (MDR), Z or XML. To support migration between metamodels defined in heterogenous modelling frameworks, EMC was extended during the development of Flock to provide a conformance checking service.

In theory then, the migration strategy described above works for both EMF (XMI 2.1) and MDR (XMI 1.2) models. In practice, we found that the way in which enumerations are encoded differs slightly between EMF and MDR, and so we had to make a slight change to the guards of the migration rules shown in Listing 1.3, producing the corresponding rules shown in Listing 1.10. In future work, we will modify the MDR driver for EMC such that the changes shown in Listing 1.10 are no longer necessary.

```
1  migrate Pseudostate to InitialNode when: original.kind.toString() = '
       pk_initial'
2  migrate Pseudostate to DecisionNode when: original.kind.toString() = '
       pk_junction'
3  migrate Pseudostate to ForkNode when: original.kind.toString() = 'pk_fork'
4  migrate Pseudostate to JoinNode when: original.kind.toString() = 'pk_join'
```

Listing 1.10. Migration using MDR enumerations

B Source Code

The following listing includes the entire Flock source code used in solving the core task.

```
1  migrate ActivityGraph to Activity {
2    var pkg := new Migrated!Package;
3    pkg.packagedElement.add(migrated);
4
5    migrated.node := original.top.subvertex.equivalent();
6    migrated.edge := original.transitions.equivalent();
7    migrated.`group` := original.partition.equivalent();
8  }
9
10 migrate Partition to ActivityPartition {
11   migrated.edges := original.contents.collect(e : Transition | e.equivalent()
         );
12   migrated.nodes := original.contents.collect(n : Original!StateVertex | n.
         equivalent());
13 }
14
15 migrate ActionState to OpaqueAction
16
17 migrate Pseudostate to InitialNode when: original.kind = Original!
       PseudostateKind#initial
18 migrate Pseudostate to DecisionNode when: original.kind = Original!
       PseudostateKind#junction
19 migrate Pseudostate to ForkNode when: original.kind = Original!
       PseudostateKind#fork
20 migrate Pseudostate to JoinNode when: original.kind = Original!
       PseudostateKind#join
21
22 migrate FinalState to ActivityFinalNode
23
24 migrate ObjectFlowState to ActivityParameterNode
25
26 migrate Transition to ObjectFlow when: original.source.isTypeOf(Original!
       ObjectFlowState) or original.target.isTypeOf(Original!ObjectFlowState)
27
28 migrate Transition to ControlFlow
29
30 migrate Guard to OpaqueExpression {
31   migrated.body.add(original.expression.body);
32 }
```

Listing 1.11. Complete Epsilon Flock listing for the core task

Model Migration with MOLA

Elina Kalnina, Audris Kalnins, Janis Iraids, Agris Sostaks, Edgars Celms

University of Latvia, IMCS, Raina bulvaris 29, LV-1459 Riga, Latvia
Elina.Kalnina@lumii.lv, Audris.Kalnins@lumii.lv, Edgars.Celms@lumii.lv,
Agris.Sostaks@lumii.lv, Janis Iraids@lumii.lv

Abstract. This paper describes the activity diagram migration from UML 1.4 to UML 2.2 in MOLA transformation language. Transformations implementing the migration task are relatively straightforward and easily inferable from the task specification. The required additional steps related to model import and export are also described.

1 Introduction

In this paper we describe the solution to Model migration case [1] for TTC 2010[1] contest, as implemented in MOLA model transformation language. The core migration task is implemented. We take as input the UML 1.4 activity models built according to the provided minimal UML 1.4 metamodel (original minimal metamodel). The target models are created according to the provided (complete) UML 2.2 metamodel.

The main point of discussion in this task is how to migrate object flows, where the best semantics preserving transformation is not so unique. We have chosen to substitute object flow states in UML 1.4 by pins in UML 2.2. This seems to be the most natural choice according to UML 2.2 specifications [2], used also in several commercial UML tools. Thus, we convert an object flow state together with incoming and outgoing transitions into a pin with incoming and outgoing object flows respectively.

We build the target model in a way that it can be imported into Eclipse with the UML 2 plug-in and visualized as a diagram, in order to check the transformation correctness more easily. This feature requires some classes to be added to the model, therefore we have chosen the complete evolved metamodel for the target, but not the minimal one.

The migration task can be implemented in MOLA in a very straightforward way once the correspondence between metamodel elements is well understood. We describe in the paper the basic principles of the solution. Before that, the situation with metamodels is described in some details, namely, how the metamodels are imported and extended and what implications this creates for source model import and target model export. The whole migration process using MOLA is briefly described as well.

[1] http://planet-research20.org/ttc2010/

2 MOLA Environment

MOLA [3] is a graphical transformation language developed at the University of Latvia. It is based on traditional concepts among transformation languages: pattern matching and rules defining how the matched pattern elements should be transformed. The formal description of MOLA and also MOLA tool can be downloaded in [4].

A MOLA program transforms an instance of a source metamodel (defined in the MOLA metamodelling language – MOLA MOF, close to EMOF) into an instance of a target metamodel.

Rule contains a declarative pattern that specifies instances of which classes must be selected and how they must be linked. Pattern in a rule is matched only once. The action part of a rule specifies which matched instances must be changed and what new instances must be created. The instances to be included in the search or to be created are specified using *class elements* in the MOLA rule. The traditional UML instance notation (instance_name:class_name) is used to identify a particular class element. Class elements may contain constraints and attribute assignments defined using simple OCL like expressions. Additionally, the rule contains association links between class elements. Class elements matched in one rule may be referenced in another one using the reference element (prefixed with ”@“ symbol).

In order to iterate through a set of the instances MOLA provides the *foreach loop* statement. The *loophead* is a special kind of a rule that is used to specify the set of instances to be iterated over. The pattern of the loophead is given using the same pattern mechanism used by an ordinary rule, but with an additional important construct. It is the *loop variable*—the class element that determines the execution of the loop. The foreach loop is executed for each distinct instance that corresponds to the loop variable and satisfies the constraints of the pattern. In fact, the loop variable plays the same role as an iterator in classical programming languages. The execution order in MOLA is specified in a way similar to UML activity diagrams.

MOLA has an Eclipse-based graphical development environment (MOLA tool [4]), incorporating all the required development support. A transformation in MOLA is compiled via the low-level transformation language L3 [5] into an executable Java code which can be run against a runtime repository containing the source model. For this case study Eclipse EMF is used as such a runtime repository, but some other repositories can be used as well (e.g. JGraLab [6]).

3 General Principles of Migration Case Solution with MOLA

The transformation development in MOLA starts with the development of metamodels. The MOLA tool has a facility for importing existing metamodels, in particular, in EMF (Ecore) format. Though MOLA metamodelling language (MOLA MOF) is very close to EMOF, and consequently Ecore, there are some issues to be solved. The current version of MOLA requires all metamodel associations to be navigable both ways (this permits to perform an efficient pattern matching using simple matching algorithms). Since a typical Ecore metamodel has many associations navigable one way, the import facility has to extend the metamodel. Another issue is

the variable coding of references to primitive data types (in this case the coding within source and target metamodels was also different).

Metamodel import facilities in MOLA are able to perform all these adjustments automatically. This way both the source (original minimal) and target (evolved) metamodels were imported into MOLA tool. Efficient transformation development in MOLA for model mapping related tasks, as the current one is, requires additional metamodel elements for storing the mapping between the source and target model elements. These elements have to be added manually. In the given case, one association between the top classes (in the inheritance hierarchy) of both metamodels is sufficient. Now the transformation itself (MOLA procedures) can be developed. The key features of transformations are described in the next section. The development ends with MOLA compilation.

Since the metamodels have been modified during import, the original source model does not conform directly to the metamodel in the repository, mainly due to added association navigability. Therefore a source model import facility is required. MOLA execution environment (MOLA runner) includes a generic model import facility, which automatically adjusts the imported model to the modified metamodel. Now the migration transformation can be run on the model. Similarly, a generic export facility automatically strips all elements of the transformed model which does not correspond to the original target metamodel. Thus a transformation result is obtained which directly conforms to the target metamodel. The transformation user is not aware of these generic import and export facilities, he directly sees the selected source model transformed.

4 MOLA Transformations for the Migration

The given migration task is very adequate for implementation in MOLA. The key issue in the transformation design is to find out which source model elements must be converted to which target model elements. The best way is to define an informal mapping, such as:

ActivityGraph/CompositeState -> Activity
ActionState -> OpaqueAction
ObjectFlowState -> Pin
Pseudostate (kind=initial) -> InitialNode
…
Transition -> ObjectFlow (when connected to ObjectFlowState)
Transition -> ControlFlow (otherwise)
and so on.

Now simple MOLA procedures can be built which directly implement the mappings. The transformation process has to be started from the top elements in the containment hierarchy, in the given case from ActivityGraph. All elements of a container, here CompositeState, are transformed using a MOLA foreach loop, running over these elements (see Fig. 1). Certainly, we assume here that an activity graph contains just one CompositeState. Further, the procedure State is just a “dispatcher”

which finds out what kind of state is really represented by the current StateVertex and invokes the corresponding transformation.

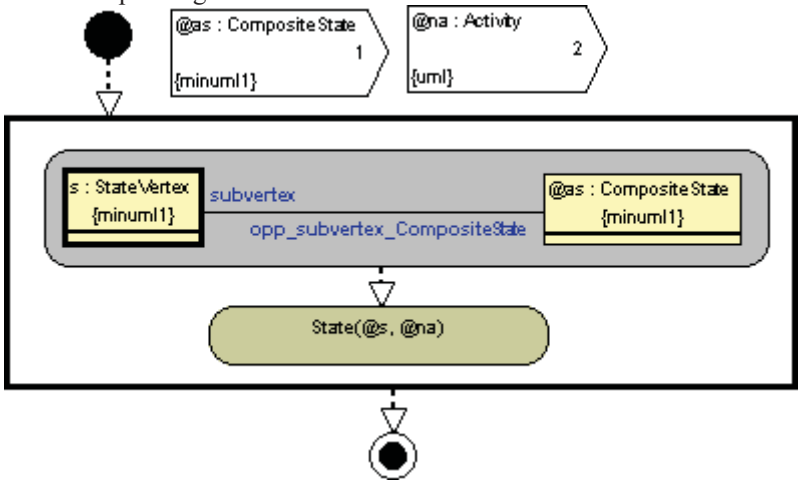


Fig. 1. MOLA loop iterating over all states in the source model

The transformation of an individual state according to the selected mapping is very straightforward, for example, the transformation of ActionState is shown in Fig. 2. Note that along with the target element (OpaqueAction) its mapping to the corresponding source element (the link sourceElement/targetElement) is built.

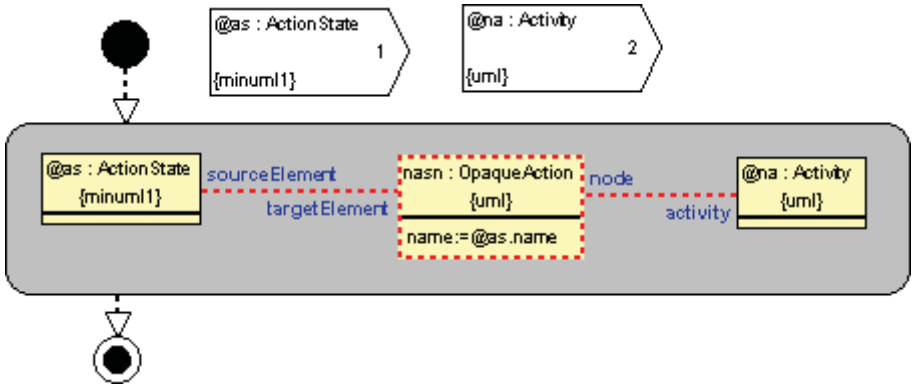


Fig. 2. Transformation of ActionState into OpaqueAction

When the nodes have been transformed, edges can be processed. Finding of end points of an edge to be created is based directly on mappings from the end points of the source edge. Finally, the partitions are created and transformed nodes are attached to the corresponding partitions (again using the mappings).

The complete set of transformation procedures is given in the appendix.

5 Conclusions

This migration case study has been very appropriate to be implemented in MOLA. The hierarchical model structure and relatively simple mappings from source to target model permitted a quite straightforward implementation by transformations in MOLA. The most complicated task was to define precisely this source-target element mapping, because of possible variations in understanding the correspondence between UML 1.4 and 2.2. Once defined, the mapping fits well into MOLA capabilities, the structure of the complete transformation given in the appendix corresponds directly to that mapping. The effort for transformation development was quite low. The required infrastructure for model management within the MOLA tool also was sufficient for the case though some less typical situations revealed a couple of deficiencies.

Certainly, a question can be raised whether such model migration task, easily to be specified by appropriate mappings, couldn't be solved more formally on the basis of these mappings. There exist some mapping based approaches (see e.g. [7, 8]) where ATL transformations can be generated by higher order transformations from a sort of mappings. However, it seems that expressiveness of these approaches could be insufficient for natural and complete specification of mappings for this case. Therefore we preferred an informal definition of the mapping with manual implementation in MOLA.

References

1. L.M. Rose, D.S. Kolovos, R.F. Paige, F.A.C. Polack: *Model Migration Case for TTC 2010*. Transformation Tool Contest 2010 Case studies, 2010 http://planet-research20.org/ttc2010/index.php?option=com_content&view=article&id=76&Itemid=131
2. OMG, Unified Modeling Language: Superstructure, version 2.2, formal/09-02-02, 2009
3. A. Kalnins, J. Barzdins, E. Celms. *Model Transformation Language MOLA*. Proceedings of MDAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62-76.
4. UL IMCS, MOLA pages, http://mola.mii.lu.lv/.
5. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs. Model Transformation Languages and Their Implementation by Bootstrapping Method. In Avron, A., Dershowitz, N., Rabinovich, A., eds.: Pillars of Computer Science. Vol. 4800, Springer LNCS, 2008, pp. 130–145.
6. Universität Koblenz-Landau, Institute for Software Technology, Graph Laboratory http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/MainResearch/Graphentechnologie/graph-laboratory-gralab
7. M. Didonet Del Fabro, P. Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. SoSyM, Vol. 8, N3, 2009, pp. 305-324, Springer-Verlag.
8. D. Lopes, S. Hammoudi, J. Bézivin and F. Jouault. Generating Transformation Definition from Mapping Specification: Application to Web Service Platform. CAiSE'05, Vol. 3520, Springer LNCS, 2005, pp. 309-325.

Appendix A: Transformation sources.

In this appendix transformation sources will be described.

In Figure 3 the main transformation procedure is demonstrated. Since in the source (minuml1) metamodel there is no Package hierarchy we simply created one “root” package in the target metamodel. To include a possibility to copy the source Package hierarchy we should change this procedure by adding a support for hierarchy.

This procedure calls the procedure Activity (see Fig. 4). It iterates through all ActivityGraphs (though most probably the source model will contain just one graph). For each ActivityGraph one UML 2.2 Activity in the root package is created. Then the transformation finds the Composite state with top association. This state contains all states of the Activity diagram. The first step is to copy all states (see Fig. 5.) The second step is to copy all Transitions (see Fig. 11.). Finally Partitions are created and states are added to appropriate Partitions (see Fig. 13.).

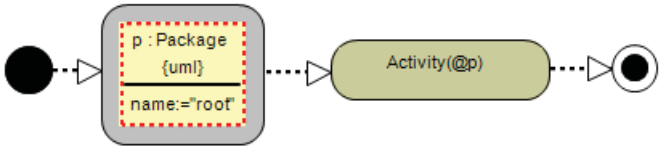


Fig. 3. Main transformation procedure. Root package is created and activity cloning is invoked.

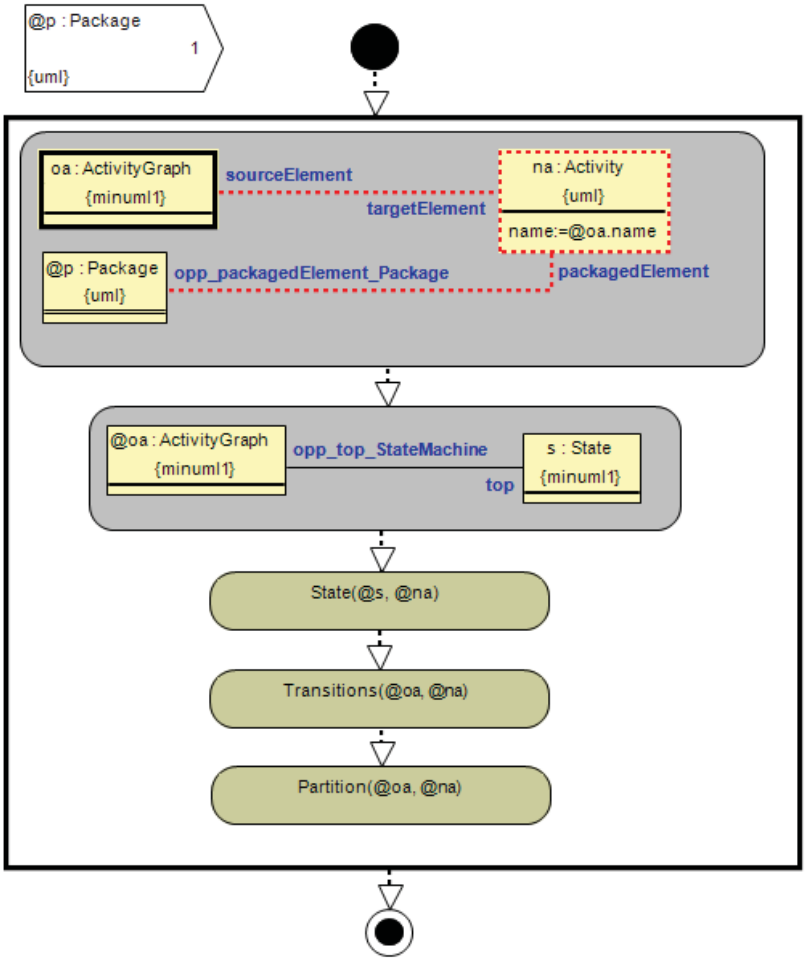


Fig. 4. Procedure Activity iterating through all ActivityGraphs in UML 1.4 and creating Activities in UML 2.2.

Copying of States is done rather special way. A general state processing mechanism is used for implementing the specified mapping between state kinds in UML 1.4 and 2.2. At first the state kind is determined (see Fig. 5.). The type of the top node is CompositeState. For it the procedure processing composite states is called (Fig. 6.). In this example it is assumed that a composite state can be used only as a top node. Therefore all nodes within a composite state are copied and placed directly in the Activity. For each node its type is determined (see Fig. 5) and the appropriate copier is called. While copying nodes the mapping association between the old and new nodes is created. The FinalState corresponds to ActivityFinalNode (Fig. 7.). The ActionState corresponds to OpaqueAction (Fig. 8.).

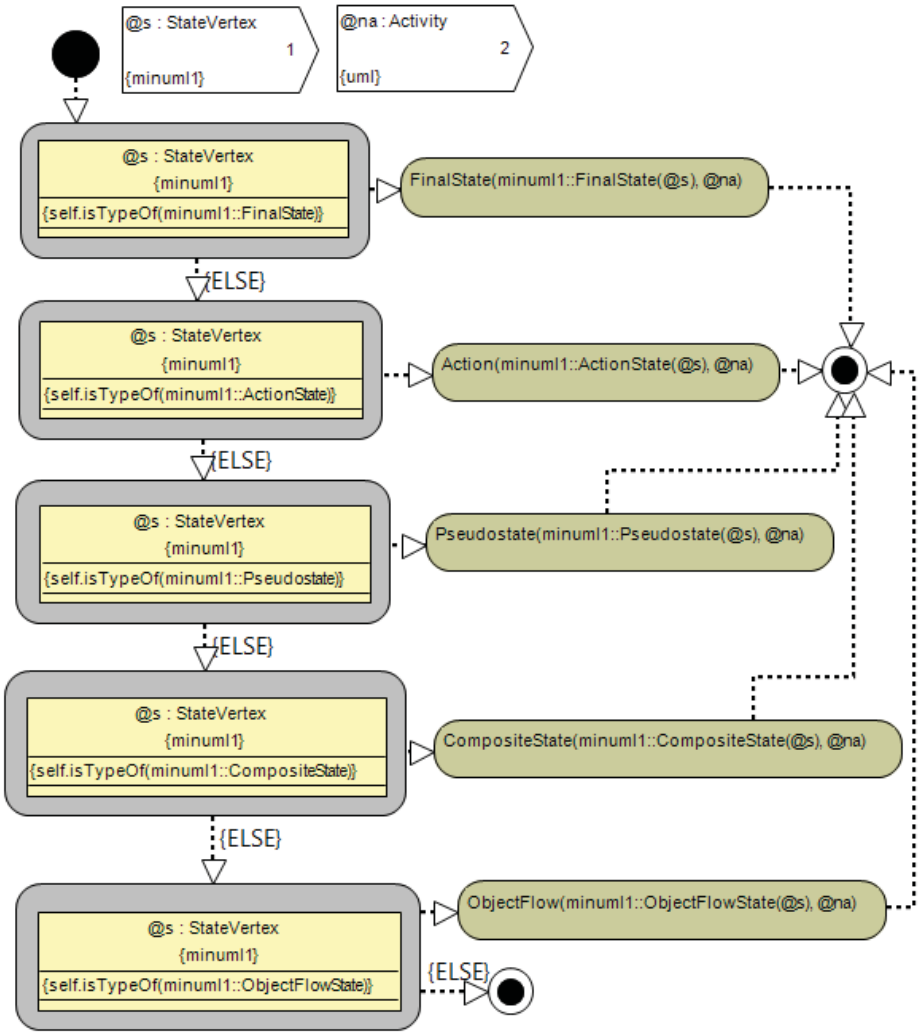


Fig. 5. Procedure State performing the state cloning. The state type is determined and the appropriate transformation is called.

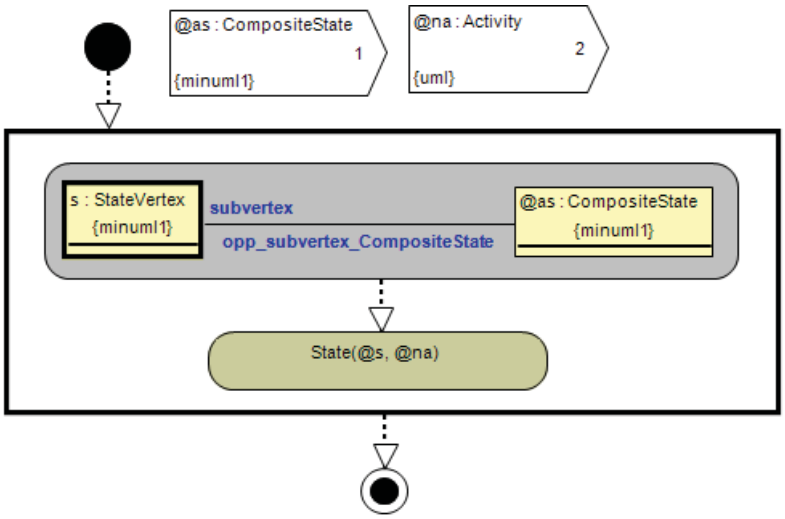


Fig. 6. Procedure CompositeState. The transformation copies all nodes contained in a Composite state and places them in the Activity.

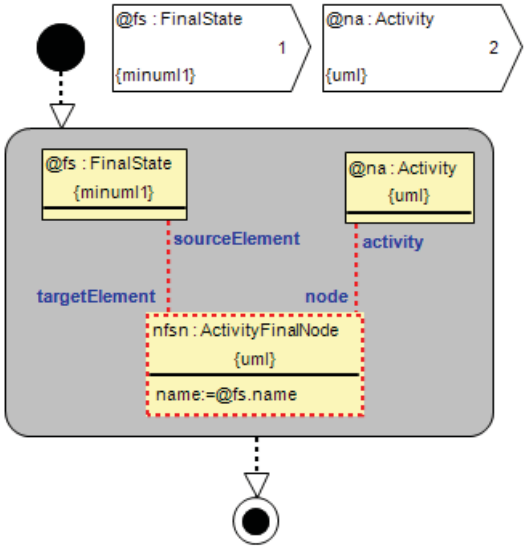


Fig. 7. Procedure FinalState transforming final state to ActivityFinalNode.

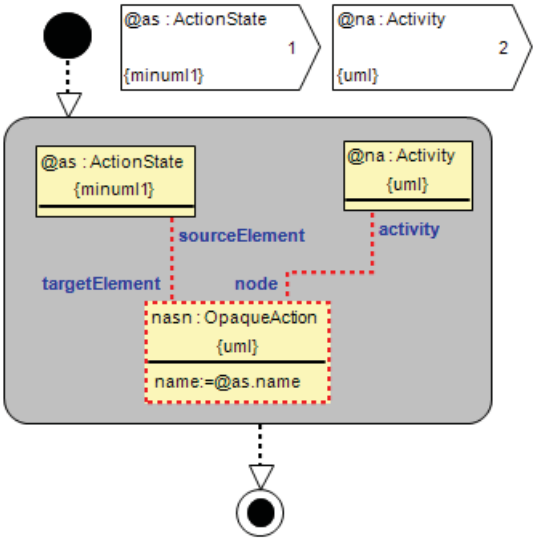


Fig. 8. Procedure Action transforming ActionState to Opaque Action.

In Figure 9 the processing of Pseudostates is described. Depending on Pseudostate kind the Node type to be created is determined. If the kind is initial an InitialNode is created. If the kind is join a JoinNode is created. If the kind is fork a ForkNode is created. If the kind is junction a DecisionNode is created. UML 2.2 has also a MergeNode but since the role of a junction is not differentiated in UML 1.4 always a DecisionNode is created for a Junction.

An ObjectFlowState is transformed to a Pin (see Fig. 10). In the updated task specification it was recommended to transform an ObjectFlowState to an ActivityParameterNode. As it was already stated in the introduction a Pin represents the semantics of ObjectFlow state more precisely. Visualization in the Eclipse UML tool is also much better.

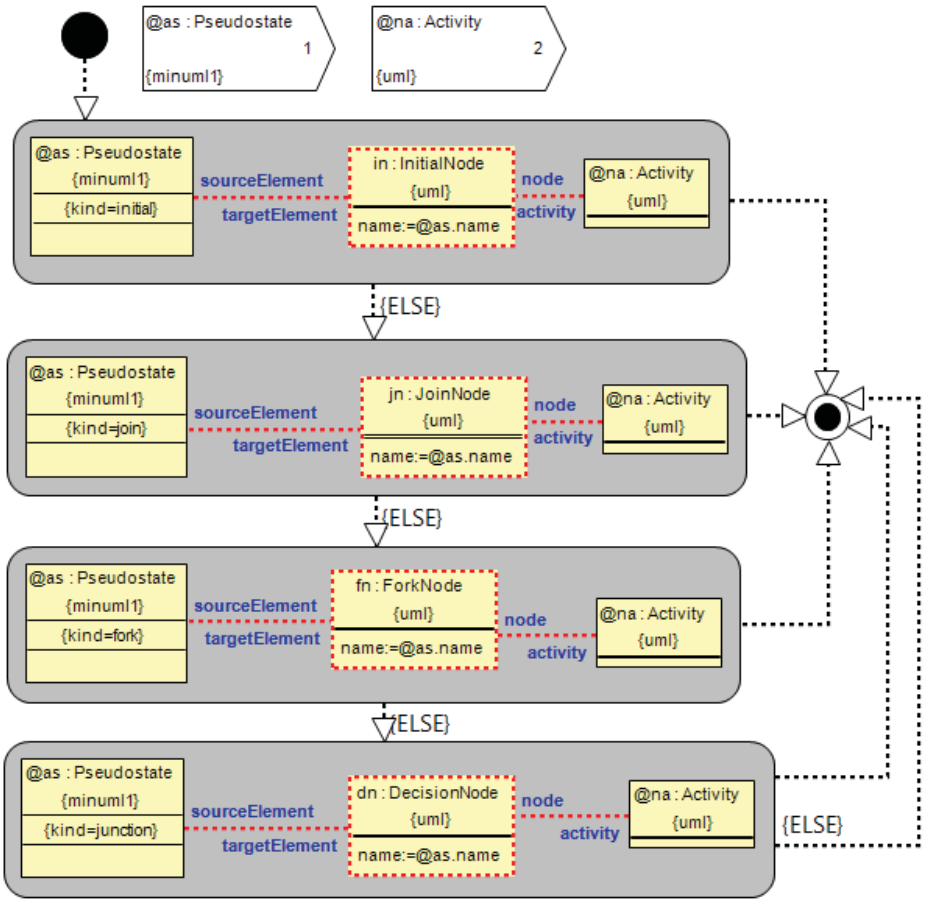


Fig. 9. Procedure PseudoState. Type of the node created depends on the Peudostate kind. If kind is initial an InitialNode is created. If kind is join a JoinNode is created. If kind is fork a ForkNode is created. If kind is junction a DecisionNode is created.

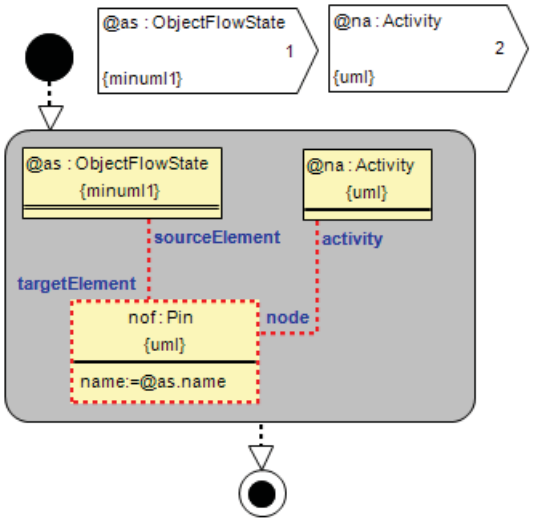


Fig. 10. Procedure ObjectFlow transforming an ObjectFlowState to a Pin.

When all states are copied the transitions are processed. The transformation iterates through all transitions in an activity graph (see Fig. 11.). It processes each transition and determines the Edge type which should be created (see Fig. 12 for transition processing). If source or target of a Transiton is an ObjectFlowState then an ObjectFlow is created. Otherwise a ControlFlow is created. Source and target nodes of the new edge are determined using the mapping association between nodes in UML 1.4 model and UML 2.2 model. The same mapping association is created between transitions and edges. Edges are also linked to the Activity.

If the transition has a guard the appropriate guard is added also to the edge in UML 2.2 (see the final rule in Fig. 12.).

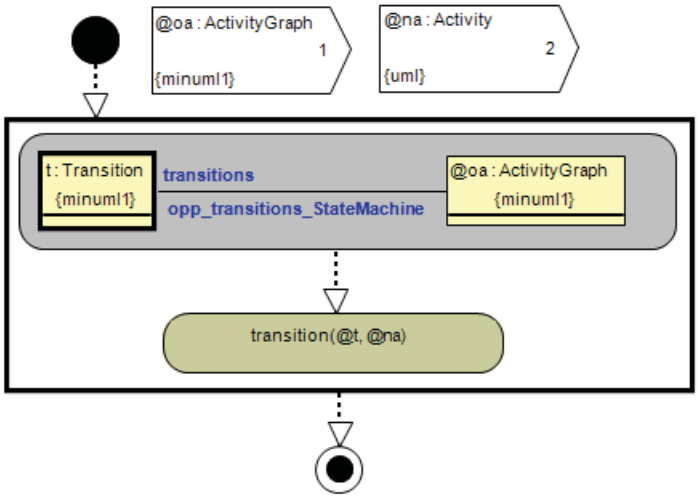


Fig. 11. Procedure Transitions finds all transitions in an ActivityGraph and processes them.

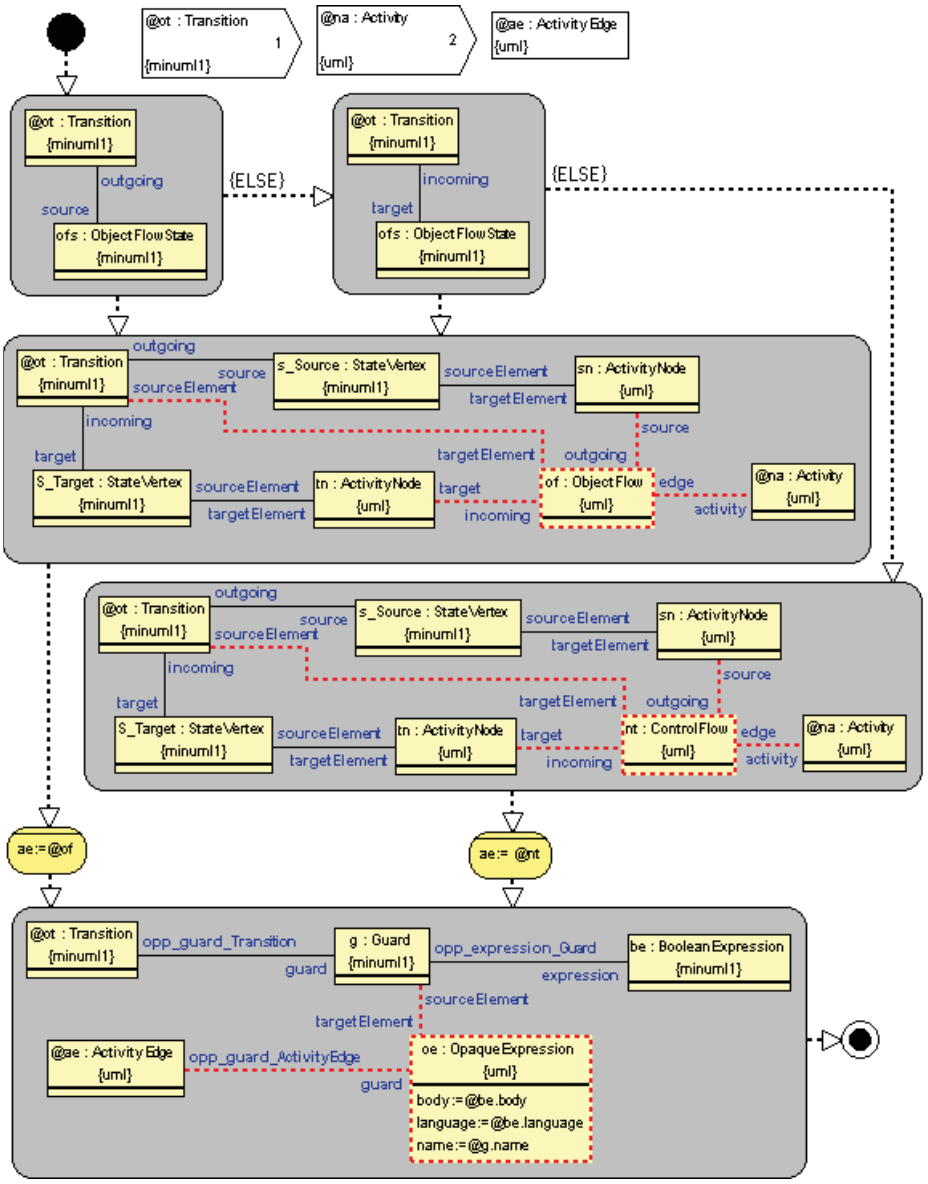


Fig. 12. Procedure transition processing one Transition. If one end of the Transition is ObjectFlow state an ObjectFlow is created. Otherwise a ControlFlow is created.

The final step of the transformation is to process partitions. The procedure in Fig. 13 iterates trough all Partitions in the ActivityGraph and creates a new ActivityPartition in the Activity for each. In Fig. 14 a loop iterates through all StateVertex (and Transitions) instances in this Partition, finds the appropriate Node (Edge) in the target model and adds it to the new ActivityPartition.

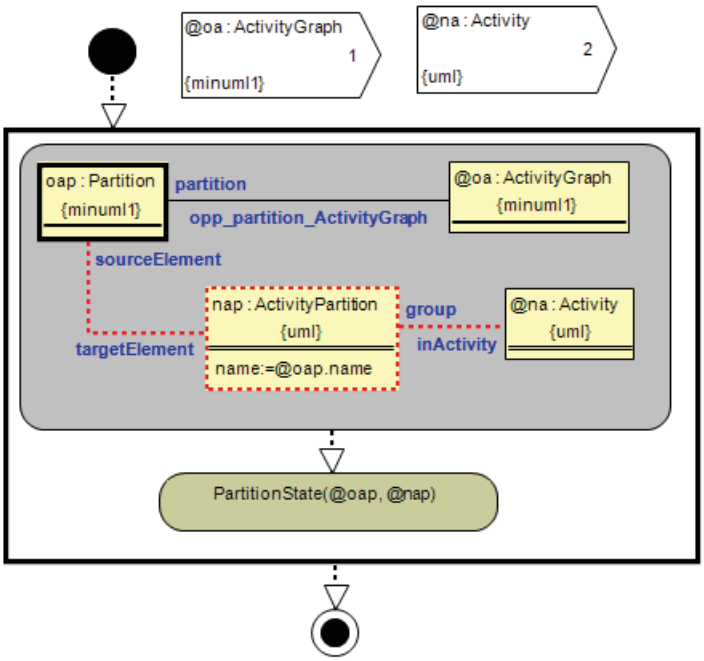


Fig. 13. Procedure Partition iterates through all partitions in the UML 1.4 model and creates an appropriate ActivityPartition in UML 2.2.

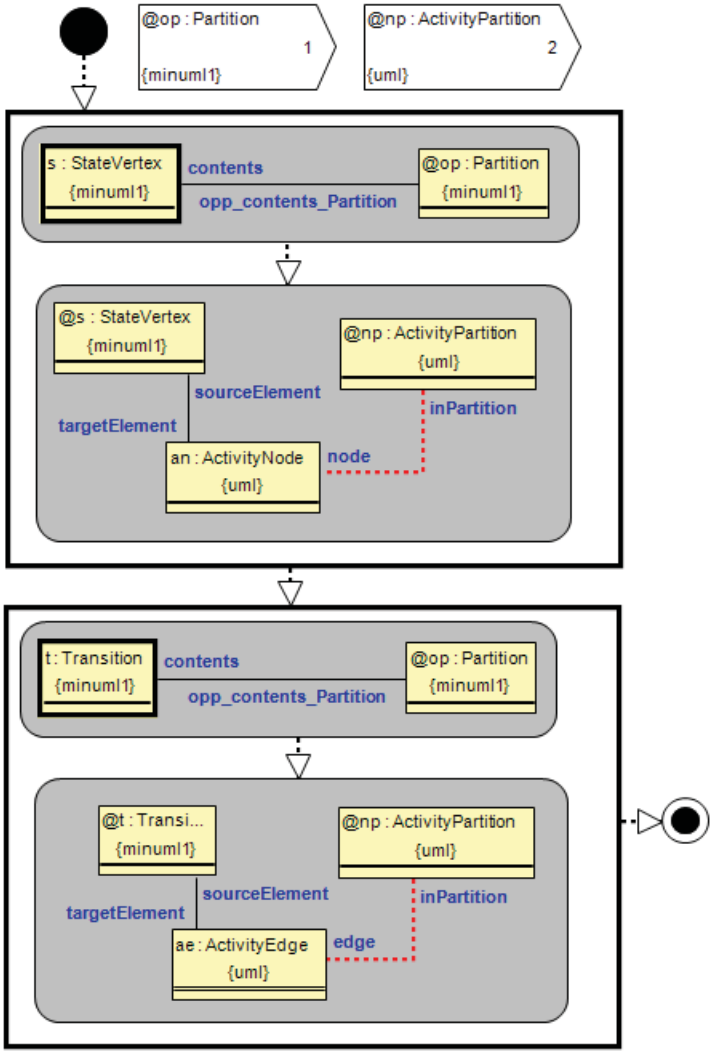


Fig. 14. Procedure PartitionState adding States and Transitions to appropriate Partitions.

UML1.4 to 2.1 Activity Diagram Model Migration with Fujaba - a Case Study

Andreas Koch
SE, Kassel University
Wilhelmshöher Allee 73
34121 Kassel
email@andreaskoch.net

Ruben Jubeh
SE, Kassel University
Wilhelmshöher Allee 73
34121 Kassel
ruben.jubeh@uni-kassel.de

Albert Zündorf
SE, Kassel University
Wilhelmshöher Allee 73
34121 Kassel
albert.zuendorf@uni-kassel.de

ABSTRACT

We have modeled a UML1.4 to UML2.2 Activity Diagram model transformation for the TTC2010 Transformation Tool Contest with the Fujaba Tool Suite. The solution uses core fujaba feature: the whole application is modeled using Story Driven Modelling [4].

1. INTRODUCTION

This paper reports on our case study with the Fujaba environment, cf. [*www.fujaba.de*], on building a UML1.4 to UML2.2 migration transformation for the TTC2010 Transformation Tool Contest.The transformation reads XMI files as input and can write a corresponding XMI2 file. The transformation is specified using Story Driven Modeling (SDM). It is similar to last years BPMN2BPEL transformation solution, see [1].

A common task in model driven software development is to migrate models when their corresponding meta models evolve. In this case study, Activity Diagrams specified in UML1.4 are migrated to the equivalent ones specified in UML2.2. This is a very exciting task, as Fujabas metamodel itself is heavily based on UML1.4 and Story Diagrams rely on activity diagram constructs. We want to migrate Fujaba to an up-to-date UML2.x metamodel in the future, so this case study will examine one possible approach to do that.

2. META MODELS AND FRAMEWORK

For getting started with the Meta Models, one solution within our Tool Suite is to import existing class binaries as .class-files (preferably available as EMF-conforming model), which would have been possible for the Eclipse UML2 plugins. As the Case Study gives .ecore-Files as reference, we imported those with the roundtrip/reverse engineering Tool UMLLab [1], which has an adapter to Fujaba. As the given models were simplified not very comprehensive, minor extensions like container classes were applied by hand. Figure 1 and 2 show the used models within the class diagram editor of Fujaba.

3. STORY DRIVEN MODELING

Story diagrams are graph rewrite rules embedded in activity diagrams that allow to query and modify the application's object graph on a high level of abstraction. Each Story diagram is associated with a method, so Story diagrams call each other by invoking other Story diagram methods. A graph rewrite rule consists of an object graph that is used as a query pattern to be matched in the current runtime object graph. It is similar to an object diagram, but adds also graph/object modifications. Our graph pattern distinguish bound objects and unbound objects. Bound objects are already matched to runtime objects and need not to be searched any more. In our notation, bound objects show only their name and omit their type. At execution time, unbound objects are matched onto runtime objects such that the overall match conforms to the search pattern graph. In addition to querying the runtime graph with a pattern graph, a graph rewrite rule also allows to model modifications of the matched elements.

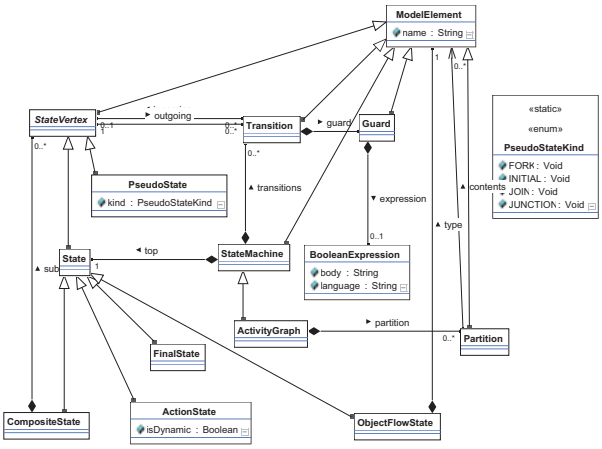


Figure 1: UML1.4 Metamodel

Figure 3 shows a story diagram method. It's the Constructor method of the class `ModelMigration` taking an `ActivityGraph` instance (from UML1.4) as parameter (see Figure 4 for associations of this class).

This method contains three graph rewrite steps. The first one looks for a state instance associated with the ActivityGraph (UML1.4) parameter, and creates, when found, an (UML2.2) `Activity` instance. Object or link creation is denoted by the ≪`create`≫ stereotype and green color. Furthermore, it assigns the same name via an attribute assignment expression and sets all required links between the

[1]http://www.umllab.de/

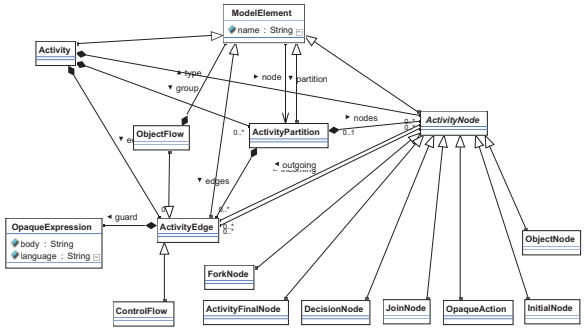


Figure 2: UML2.2 Metamodel

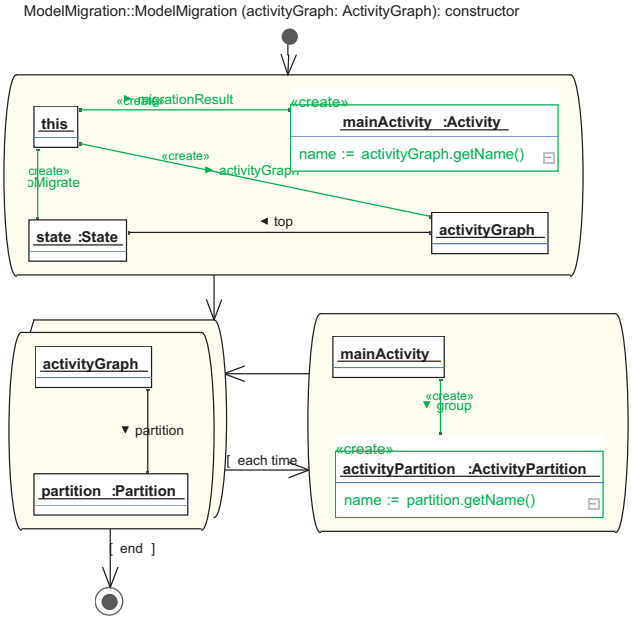


Figure 3: migration entry point: the constructor

migration and the other object. This is a preparation step of the migration transformation, creating the parent instance (Activity) of the migrated model. The second rewrite rule just looks for all partitions associated with the input activity graph and creates, for each found (that is indicated by the double outer box) partition, an `ActivityPartition` instance (UML2.2).

Graph rewrite rules are an excellent means to model an applications behavior in terms of operations on its underlying object graph. The graph patterns allow to express complex graph queries that are (with some exercise) easy to read and to understand. Thereby, the programs are easier to extend and to maintain.

4. THE TRANSFORMATION

Figure 4 shows the class diagram of our transformation. The central class `ModelMigration` refers to both the input graph as UML1.4 `AcivityGraph` and the output graph ref-

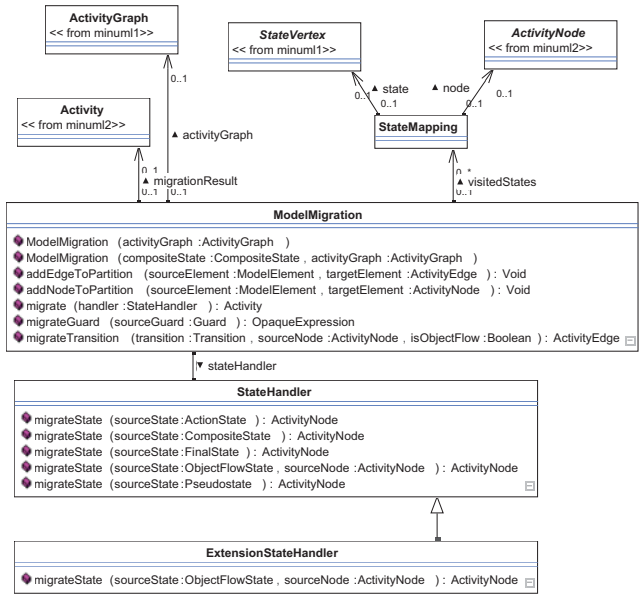


Figure 4: Migration Engine class diagram

erenced by it's parent class `Activity`. To distinguish between the different operation modes, as required by the extension discussed in 5.1, it delegates some migration work to a `StateHandler`. Furthermore, it remembers already migrated states via a `StateMapping` pointing to elements of both meta models.

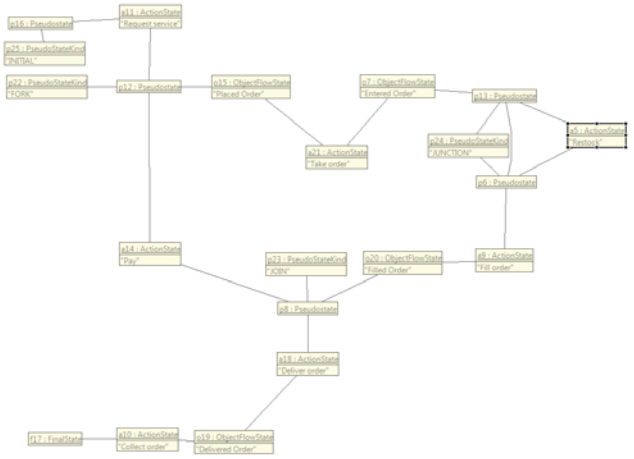


Figure 5: Sample input model/graph as eDOBS view

Figure 5 shows the input model, either constructed as separate graph construction rule or imported via XML1.2 file, as object model in the heap memory, visualized by the eDOBS runtime debugger tool. Figure 6 in the appendix shows the same model with partitions, equal to the model given in the case study. In the following, we will show some of the methods performing the migration transformation, and the corresponding runtime object graphs. We start the transformation by calling `migrate(StateHandler)` (cf. figure 7) on a model migration instance. As a result of it's constructor

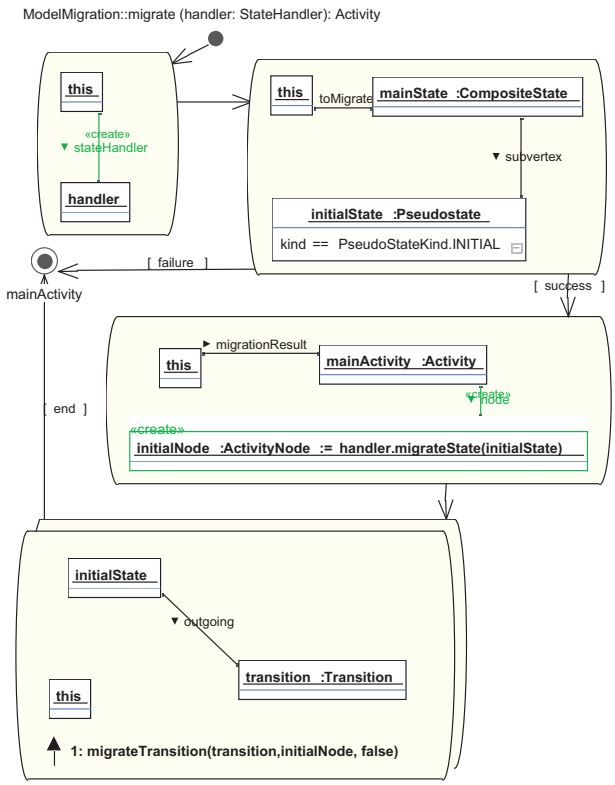


Figure 7: main migration method: find initial node and continue

method, it is already connected to an Activity instance. Figure 7 shows the story diagram of this method. It searches for the Pseudostate declared as initial state and creates a corresponding UML2.2 `InitialNode` instance. The transformation continues after that by invoking another story pattern: `migrateTransition(trans, node, boolean)` is called via a collaboration statement on the `ModelMigration` instance itself (this).

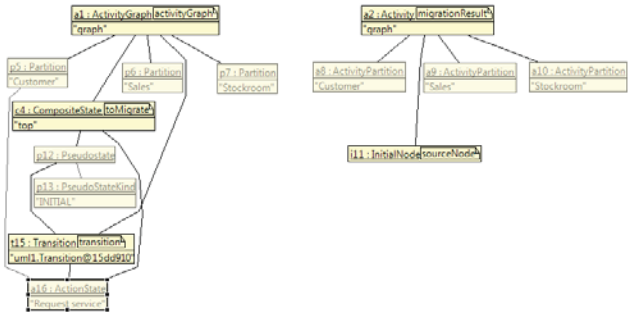


Figure 8: Runtime objects during transformation: first recursion

The `migrateTransition(trans, node, boolean)` method is shown in Figure 10. Figure 8 shows an excerpt of the object graph for the sample model just before the method gets executed the first time. It will now transform the Actionstate *a16*.

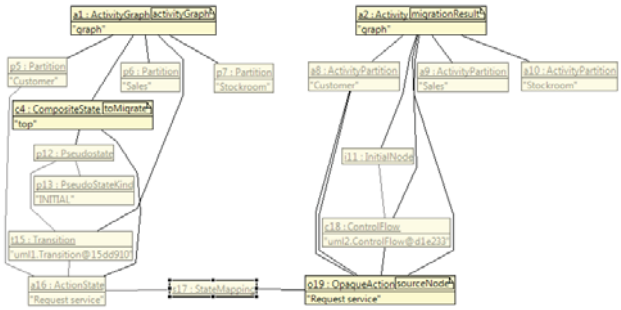


Figure 9: Runtime objects during transformation: second recursion

As a16 still has successor(s) (cf. figure 5), the algorithm will continue and transform the next action state *a11*. Figure 9 shows the object graph when entering `migrateTransition(trans, node, boolean)` the second time. A state mapping remembers that a16 has been transformed already.

In the following, the input graph is now traversed following all transitions between state vertices in a depth-first manner. We remember already transformed states with state mapping instances, so a path joining an already traversed path will terminate the transformation for that branch. Of course, this approach relies on valid input models with exactly one initial node, but checking the input model beforehand was out of scope of this work.

Figure 10 shows the central transformation rule: the left part determines the concrete instance type of the state and calls the corresponding `migrateState(...)` method of the handler. Furthermore, the method creates the additional flow instance and links it to it's source and target node. It converts also the transition guard. Finally it creates a state mapping. Transformations for the individual UML1.4 state types are handled in separate methods of the handler. As an example, figure 11 shows the transformation of an object flow.

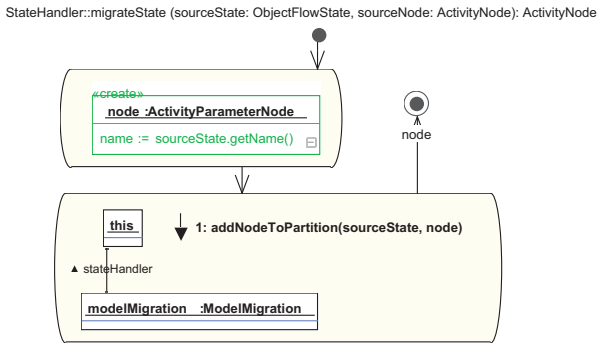


Figure 11: migrate an object flow state

Figure 12 shows a more complex graph rewrite rule: It's responsible for migrating the different Pseudostate kinds to a corresponding activity node.

5. EXTENSIONS

5.1 Alternative Object Flow State Migration

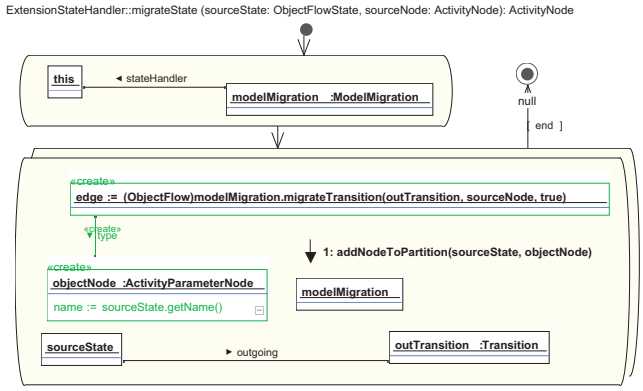


Figure 13: alternative migration of an object flow state

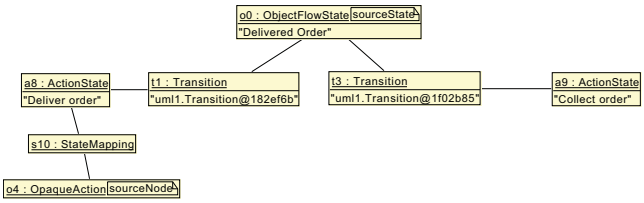


Figure 14: Object graph before an alternative extended migration

In contrast to the behavior specified in the `StateHandler.migrateState(ObjectFlow...)` method (cf. figure 11), our algorithm can also be run with the `ExtendedStateHandler`, which overrides this method and specifies a different transformation: Figure 13 shows the Story diagram of the alternative method. Figure 14 shows the object graph surrounding the "Delivered Order" object flow state. When executing the alternative transformation, *o0* is mapped to the sourceState 'variable'. Its predecessor is the action state *a8*. The object graph shows that this state has been migrated already, indicated by the state mapping *s10* to *o4*, the corresponding source UML2.2 opaque action. In this method, a new object flow edge instance gets created. The method skips the transition *t1* by invoking `migrateTransition(trans, node, true)` directly, so the algorithm will continue with transition *t3*. Furthermore, the edge gets an new activity parameter node as type, preserving the source state's name. The result of this method can be seen in figure 15. As you can see by the mapping links, that the 'middle' object *o0* is migrated by a typed object flow edge.

5.2 XMI Import and Export

Importing and exporting XML was implemented by hand using the JDOM library. The import/export code adapts to the Fujaba generated model elements and is as small as about 100 LOC each. Besides these classes, everything else was modeled using Fujaba Story diagrams. The Importer for XMI1.2 and XMI2.0 uses a reflective mechanism and is independent of the meta model to be loaded, although schema checking etc. has not been implemented. By using reflective mechanisms, it is able in instantiate both UML1.4 and UML2.2 models without introducing a compile time dependency to those. The `XMLExporter` simply traverses the

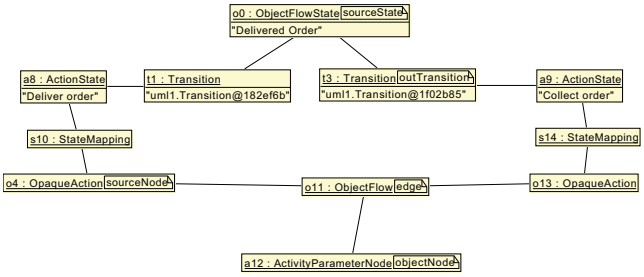


Figure 15: Object graph after an alternative extended migration

activity graph structure and writes the corresponding XML tree to disk. See the online solution via SHARE [3] for full listing of the XMI import/export source code.

6. SUMMARY AND OUTLOOK

This case study was a nice exercise for our Fujaba Case Tool. The transformation rules are intuitive and the iterative approach is easy to understand. Our approach reduces the problem down to eight graph transformation rules (not counting helper methods), which take an input structure fragment and create the corresponding output. The Importer and additional object graph creators show that the migration transformation works correctly. The development of the transformation was greatly supported by the eDOBS [2] eclipse plugin in combination with the Design Level Debugging functionality, which visualizes the runtime object graphs, so one can follow the execution of the transformation in detail by stepping with the debugger and watch how the graph is modified.

7. REFERENCES

[1] R. Jubeh. BPMN to BPEL transformation with Fujaba - a Case Study. In *GraBaTs 2009, 5th International Workshop on Graph-Based Tools*, Zurich, Switzerland, 2009.

[2] The EDobs Dynamic Object Browser. http://www.se.eecs.uni-kassel.de/typo3/index.php?edobs, 2006.

[3] Fujaba Model Migration SHARE Online Solution. http://is.tm.tue.nl/staff/pvgorp/share/, 2010.

[4] A. Zündorf. Rigorous object oriented software development. Habilitation Thesis, University of Paderborn, 2001.

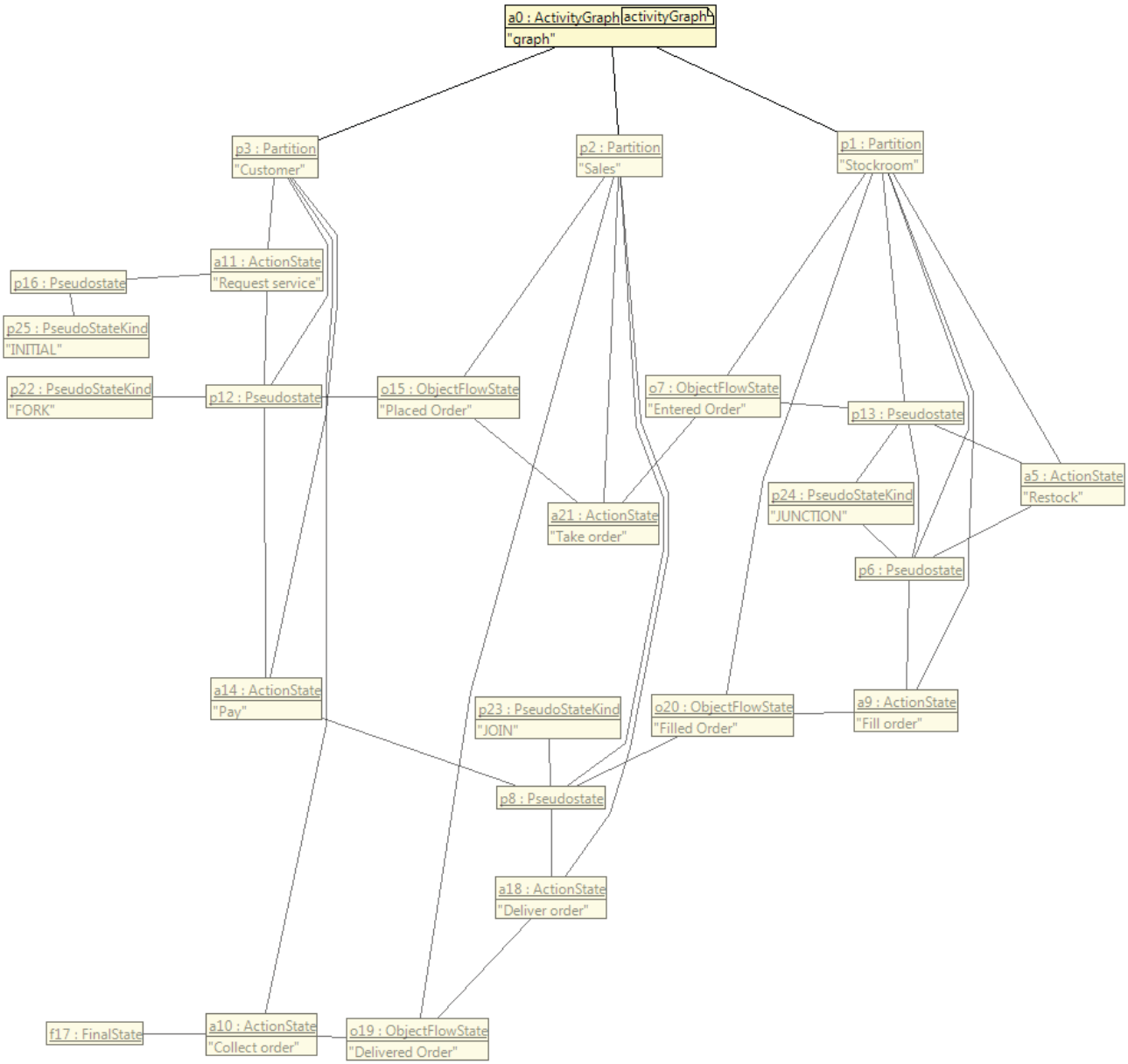


Figure 6: Sample input model/graph as eDOBS view

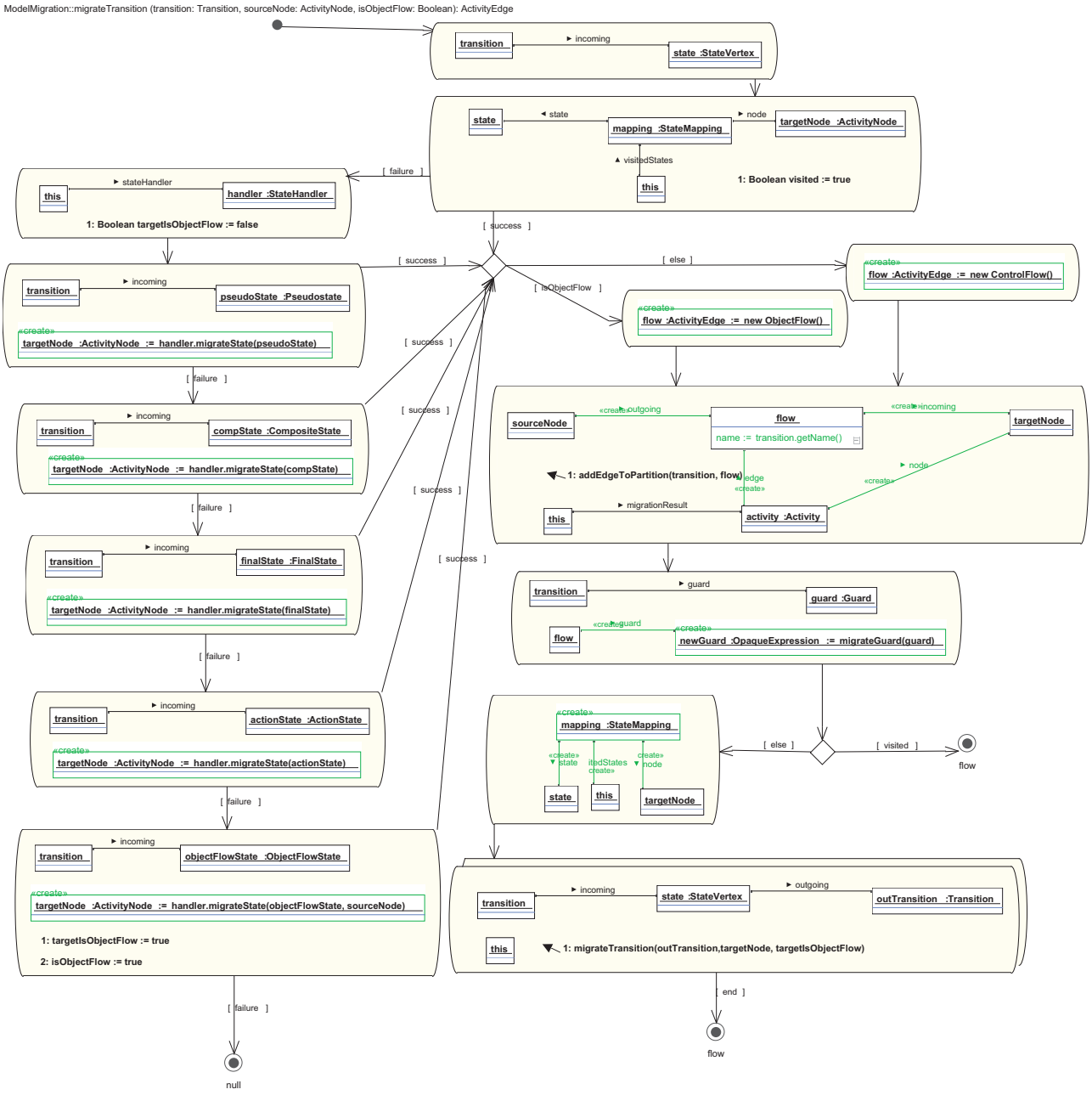


Figure 10: Migration transition method

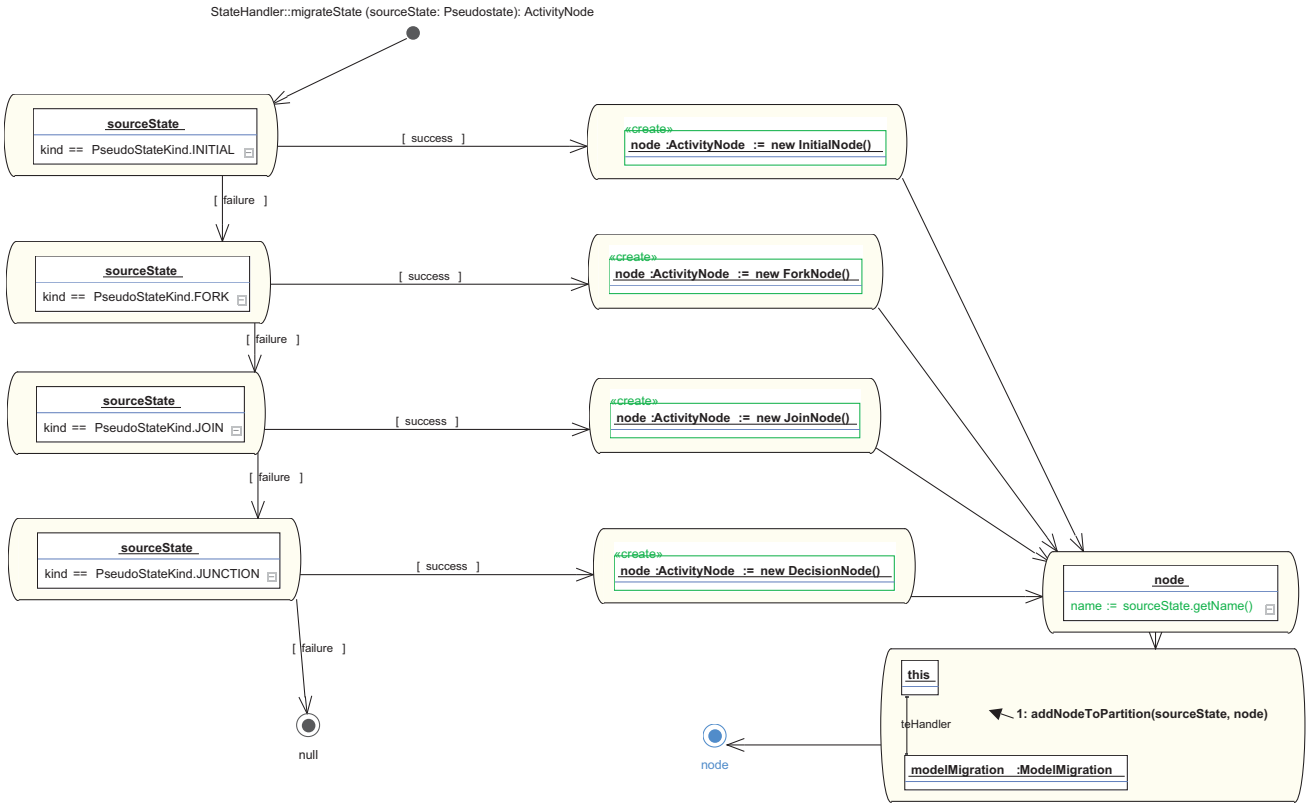


Figure 12: Migration of an pseudo state: mapping an enum to different instances

A GrGen.NET solution of the Model Migration Case for the Transformation Tool Contest 2010

Sebastian Buchwald Edgar Jakumeit

June 3, 2010

1 Introduction

The challenge of the Model Migration Case [1] is to migrate an activity diagram from UML 1.4 to UML 2.2. We employ the general purpose graph rewrite system GrGen.NET (www.grgen.net) to tackle this task. After a short description of the GrGen.NET system, we give an introduction into our solution of the core assignment, followed by a discussion of the various extensions proposed by the case authors. Finally, we conclude.

2 What is GrGen.NET?

GrGen.NET is an application domain neutral graph rewrite system developed at the IPD Goos of Universität Karlsruhe (TH), Germany [2]. The feature highlights of GrGen.NET regarding practical relevance are:

Fully Featured Meta Model: GrGen.NET uses attributed and typed multigraphs with multiple inheritance on node/edge types. Attributes may be typed with one of several basic types, user defined enums, or generic set and map types.

Expressive Rules, Fast Execution: The expressive and easy to learn rule specification language allows straightforward formulation of even complex problems, with an optimized implementation yielding high execution speed at modest memory consumption.

Programmed Rule Application: GrGen.NET supports a high-level rule application control language, Graph Rewrite Sequences (GRS), offering logical, sequential and iterative control plus variables and storages for the communication of processing locations between rules.

Graphical Debugging: GrShell, GrGen.NET's command line shell, offers interactive execution of rules, visualising together with yComp the current graph and the rewrite process. This way you can see what the graph looks like at a given step of a complex transformation and develop the next step accordingly. Or you can debug your rules.

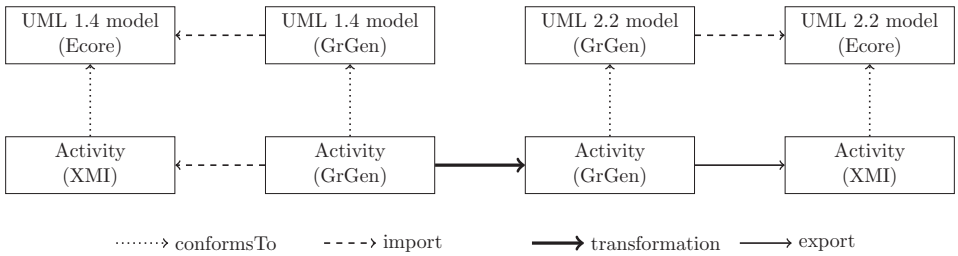


Figure 1: Processing steps of the model migration. The transformation and the XMI export are written in GrGen.NET languages. Import is handled by a supplied import filter, which generates .gm files as an intermediate step.

3 The Core Assignment

The task of the core assignment is to migrate an activity diagram conforming to an UML 1.4 metamodel to a semantically equivalent activity diagram conforming to an UML 2.2 metamodel. The aim of the task is to evaluate the solutions regarding correctness, conciseness and clarity, or better to evaluate the participating tools in how far they allow for such solutions. Before the transformation can take place, the activity diagram needs to be imported from an Ecore file describing the source model and an XMI file specifying the graph. Afterwards the resulting activity diagram has to be exported into an XMI file conforming to a given Ecore file describing the target model.

3.1 Importing the Graph

As GrGen.NET is a general purpose graph rewrite system and not a model transformation tool, we do not support importing Ecore metamodels directly (we directly support GXL and GRS files). Instead we offer an import filter generating an equivalent GrGen.NET-specific graph model (.gm file) from a given Ecore file by mapping classes to GrGen node classes, their attributes to corresponding GrGen attributes, and their references to GrGen edge classes. Inheritance is transferred one-to-one, and enumerations are mapped to GrGen enums. Class names are prefixed by the names of the packages they are contained in to prevent name clashes. Afterwards the instance graph XMI adhering to the Ecore model thus adhering to the just generated equivalent GrGen graph model is imported by the filter into the system to serve as the host graph for the following transformations. The entire process is shown in Figure 1 above.

3.2 Transformation

The transformation is done in several passes, one pass for each node type and edge type, respectively. Each pass consists of the iterated application

of one single rule which matches a node or edge of an UML 1.4 type to process and rewrites it to its corresponding UML 2.2 target type. Instead of handling the edges together with the nodes we process them separately. This relieves us from taking care of the multiplicities of the incoming or outgoing edges, which allows for a simple solution built from very simple rules. This approach is possible because of the availability of a retype operator which allows to change the type of a node (edge) keeping all incident edges (nodes).

We start the detailed presentation of the solution with the following example rule:

```
rule transform_ActionState {
    state:minuml1_ActionState;

    modify {
        opaque:uml_OpaqueAction<state>;

        eval { opaque._name = state._name; }
    }
}
```

Rules in GrGen consist of a pattern part specifying the graph pattern to match and a nested rewrite part specifying the changes to be made. The pattern part is built up of node and edge declarations or references with an intuitive[1] syntax: Nodes are declared by `n:t`, where `n` is an optional node identifier, and `t` its type. An edge `e` with source `x` and target `y` is declared by `x -e:t-> y`, whereas `-->` introduces an anonymous edge of type `Edge`. Nodes and edges are referenced outside their declaration by `n` and `-e->`, respectively. The rewrite part is specified by a `modify` block nested within the rule. Usually, here you would add new graph elements or delete old ones, but in this case we only want to retype them (also known as relabeling in the graph rewriting community). Retyping is specified with the syntax `y:t<x>`: this defines `y` to be a retyped version of the original node `x`, retyped to the new type `t`; for edges the syntax is `-y:t<x>->`. These and a lot more language elements are described in more detail in the extensive GrGen.NET user manual [2].

In the example a node `state` of type `ActionState` (mind the package name mangling) is specified in the pattern part to get matched. In the rewrite part it is specified to get retyped to a node `opaque` of type `OpaqueAction`. Furthermore the `name` attribute of the original node is assigned to the `name` attribute of the new, retyped node in the attribute evaluation `eval`.

[1] it was used in the discussion forum for this case as textual notation to describe the patterns

Most of the rules are as easy as this one. Only for a few types, the rewriting additionally depends on further local information or the context where the graph element appears in. An example for further local information to be taken into account is the rule for the transformation of the `Pseudostate` nodes. An `alternative` construct is used here (namespace prefixes were removed due to space constraints):

```
rule transform_PseudoState {
    state:Pseudostate;

    alternative {
      Initial {
        if { state._kind == PseudostateKind::_initial; }
        modify {
            initial:InitialNode<state>;
            eval { initial._name = state._name; }
        }
      }
      Join {
        if { state._kind == PseudostateKind::_join; }
        modify {
            join:JoinNode<state>;
            eval { join._name = state._name; }
        }
      }
      Fork { /* similar to the cases above */ }
      Junction { /* similar to the cases above */ }
    }

    modify {
    }
}
```

There are four possible target types for the source type, so four different alternative cases are specified, each relabeling to one of the types in their nested rewrite part. The correct type depends on the kind value of the source node; this condition is checked by an attribute condition given within the `if`-clause (fitting to the rewrite).

An example for context dependency is the rewriting of the `Transition` nodes. If they are linked to a node of `ObjectFlowState` (rewritten to `Pin`), they get retyped to nodes of type `ObjectFlow`, otherwise to nodes of type `ControlFlow`. This is expressed again with an `alternative` statement as you can see below, with a case for the transformation to control flow, preventing by negative patterns that it matches on the object flow situation, and two almost identical cases for the incoming and outgoing object flow.

```
rule transform_Transition {
    transition:Transition;

    alternative {
      controlFlow {
        negative {
           transition <-:StateVertex_incoming- :uml_Pin;
        }
        negative {
           transition <-:StateVertex_outgoing- :uml_Pin;
        }
        modify {
            cf:uml_ControlFlow<transition>;
            eval { cf._name = transition._name; }
        }
      }
      incomingObjectFlow {
        transition <-:StateVertex_incoming- :uml_Pin;
        modify {
            of:uml_ObjectFlow<transition>;
            eval { of._name = transition._name; }
        }
      }
      outgoingObjectFlow { /* similar to case above */ }
    }

    modify {
    }
}
```

As a final example for the transformation core let us have a look at one of the rules for the retyping of the edges – they follow the GrGen design target of handling nodes and edges as uniform as possible: they are nearly identical to the rules for the retyping of the nodes:

```
rule transform_StateMachine_transitions {
    -e:minuml1_StateMachine_transitions->;

    modify {
        -:uml_Activity_edge<e>->;
    }
}
```

The four shown rules are applied from within the graph rewrite script for the core solution executed by the GrShell:

```
import original_minimal_metamodel.ecore
       evolved_metamodel.ecore
       original_model.xmi core.grg
# Transform nodes
xgrs ... | transform_ActionState* | transform_PseudoState*
         | transform_Transition* | ...
# Transform edges
debug xgrs ... | transform_StateMachine_transitions* | ...
```

The `xgrs` keyword starts an extended graph rewrite sequence, which is the rule application control language of GrGen (prepending `debug` before `xgrs` allows you to debug the sequence execution in GrShell). The single rules are applied iteratively by the star operator until no match is found. They are linked by eager ors which get executed from left to right, yielding the disjunction of the truth values emanating from iteration execution (a rule which can get applied because a match is found in the graph succeeds(`true`), whereas a rule for which no match is found fails(`false`); the star operator always succeeds).

Overall our solution complies to the following scheme:

- For each node or edge type there is one rule relabeling an element of this type, often containing nothing more than this relabeling, sometimes using alternatives to decide between possible target types depending on the context.
- Each of this rules gets applied exhaustively, one rule after the other; first handling all node types, then handling all edge types (thus a few context dependent rules match against nodes/edges of types from the source and target model).

The modulare nature of this approach facilitates extensions regarding the support of additional UML elements and the realization of alternative semantics (see section 4). (Please note that it would be possible to shrink the number of rules down to one to be applied iteratively using an alternative with a lot of cases; or by using strings instead of types to encode the model types, transforming them by string replacement using map types and map lookup. But we prefer to give straight forward real-world solutions.)

3.3 Exporting the Graph

Our XMI exporter consists of several graph transformation rules that traverse the graph hierarchically while emitting the corresponding XMI tags. The following rule exports an activity:

```
rule emit_activity {
  activity:uml_Activity;
  activity -:DumpEdge-> d:DumpNode;

  modify {
    emit("  <packagedElement xmi:type=\"uml:Activity\"");
    emit(" xmi:id=\"" + d.id + "\"");
    emit(" name=\"" + activity._name + "\">\n");
    exec(emit_activity_nodes(activity));
    exec(emit_activity_edges(activity));
    exec(emit_activity_groups(activity));
    emit("  </packagedElement>\n");
  }
}
```

Since we need a unique ID for each node, we connect each node to a `DumpNode` node that provides such an ID. The `emit` statements emit the given strings; here they fill up the template of the XMI tag with node attributes. The rules executed by the `exec` statement are responsible for emitting all nodes, edges, and groups of the current activity, respectively.

4 The Extensions

In addition to the core task three extensions were proposed:

- Alternative Object Flow State Migration Semantics
- Concrete Syntax
- XMI

We will discuss them and our solutions to them in the following sections.

4.1 Alternative Object Flow State Migration Semantics

The goal of this extension is to transform a node of type `ObjectFlowState` linked to nodes of type `Transition` to a node of type `ObjectFlow` only, instead of transforming them to a node of type `Pin` linked to nodes of type `ObjectFlow`. The purpose of this task is to evaluate how well the transformation tools can cope with transformations which do not map source pattern elements injectively to target pattern elements. Or to put it in another way: require real graph rewriting instead of only graph relabeling. As GrGen.NET is a graph *rewrite* system in the first place, this does not cause any problems:

```
rule transform_ObjectFlowState2 {
  state:ObjectFlowState;
  s1:StateVertex <-:Transition_source- t1:Transition;
  t1 -:Transition_target-> state;
  state <-:Transition_source- t2:Transition;
  -:Transition_target-> s2:StateVertex;
  state <-:Partition_contents- p:Partition;

  modify {
    delete(state, t2);

    flow:ObjectFlow<t1>;
    flow -:ActivityEdge_target-> s2;
    flow <-:ActivityNode_incoming- s2;
    flow -:ActivityEdge_inPartition-> p;
    flow <-:ActivityPartition_edge- p;

    eval { flow._name = state._name; }
  }
}
```

4.2 Concrete Syntax

The goal of this extension is to transform the concrete syntax, i.e. the user drawn diagram layout, from the given diagram to the concrete syntax of the tool under consideration. As GrGen.NET was originally developed for handling compiler intermediate language graphs which do not possess a user drawn layout we do not offer a concrete visual syntax showing user editing. So we cannot transform any concrete syntax — but as the ultimate goal of a concrete syntax is a nice layout, we want to present a solution of a different kind: we offer a highly customizable graph viewer with automatic layout – which comes near to the concrete syntax of the activity diagram given as you may see in Figure 2.

In a lot of situations an automatic layout is the better choice as it delivers a very nice presentation without user interaction (besides a few lines of configuration code). You can configure the graph viewer named yComp which gets executed from GrShell to use one of several available layout algorithms – with hierarchic, compilergraph and organic being the most useful ones. You can configure for every available node or edge type in which colour with what node shape or edge style it should be shown, with what attribute values or fixed text as element labels or tags it is to be displayed, or if it should be shown at all. Further on you can configure graph nesting by registering edges at certain nodes to define a containment hierarchy, causing the nodes to become displayed as subgraphs containing

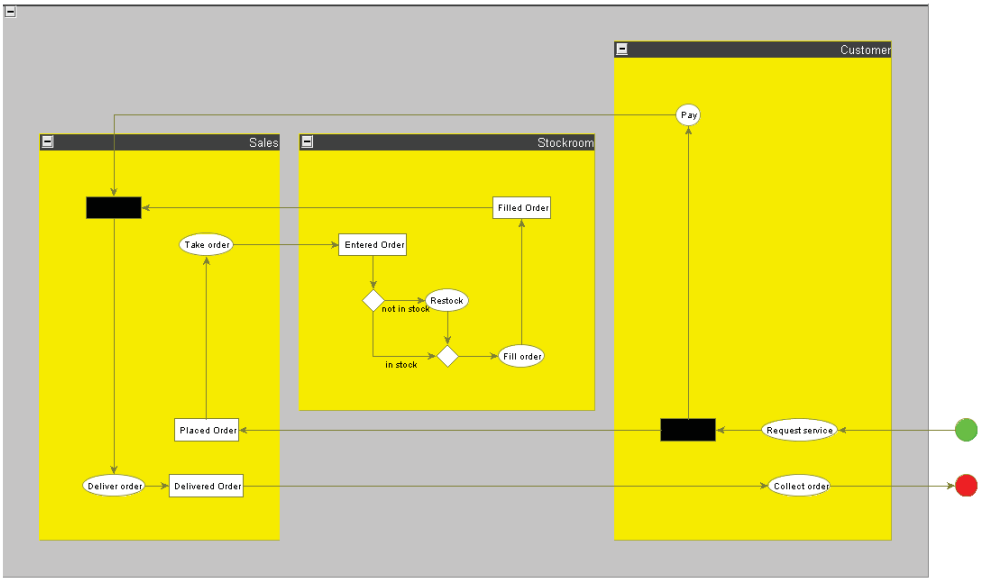


Figure 2: The transformed UML 2.2 activity diagram as displayed by yComp

the elements to which they are linked by the given edges. Additionally it offers automatic cutting of hierarchy crossing edges, marking the begin and end by fat dots, allowing to jump to either one by clicking on the other. You can easily define a layout matching your graph class by a few dozen lines of configuration information and afterwards you get a fully automatic, high-quality layout of your instance graphs.

In the following listing, we show an excerpt from our configuration file for customizing the graph layout of the UML 2.2 diagram (the first two lines ensure that each `ActivityPartition` node contains all nodes that are connected by an outgoing `uml_ActivityPartition_node` edge):

```
dump add node uml_ActivityPartition group by
         hidden outgoing uml_ActivityPartition_node
dump set node uml_ActivityPartition labels off
dump add node uml_ActivityPartition shortinfotag _name

dump add edge uml_ActivityNode_inPartition exclude

dump set node uml_DecisionNode shape rhomb
dump set node uml_DecisionNode labels off
dump set node uml_DecisionNode color white
```

4.3 XMI

The goal of this extension is to import an activity diagram given in XMI 1.x instead of XMI 2.x. The task is to write an import filter for an outdated format used in the model transformation community. While we did write an import filter for XMI 2.x (a slightly extended version of the filter originally introduced for the GraBaTs 2009 Reverse Engineering case [3]), we will not write a filter for XMI 1.x. The XMI 2.x filter allows to use the transformation capabilities of GrGen.NET with data in the Ecore/XMI format common to the model transformation community briding the graph rewriting and the model transformation communities; but supplying another Ecore/XMI filter just for the sake of this contest is beyond our scope. And we think it is out of scope even for the TTC as such: we doubt writing an import filter is a worthwhile challenge for a transformation tool contest comparing the transformation capabilities of the competing tools in order to foster the progress in software engineering. If it really were, we would like to propose to the authors from the model transformation community to follow our example bridging both worlds by writing an import filter for GXL, the standard in the graph rewriting community.

5 Conclusion

In this paper we presented a GrGen.NET solution to the Model Migration challenge of the Transformation Tool Contest 2010. The activity diagram conforming to the UML 1.4 metamodel was imported by an import filter under remapping to the graph concepts supported by GrGen. It was transformed to a semantically equivalent activity diagram conforming to an UML 2.2 metamodel using graph *relabeling*: this ability of retyping nodes (edges) while keeping their incident edges (nodes) allowed us to give a very concise and simple solution to the core task of the Model Migration challenge, exhaustively relabeling nodes then edges with very simple rules until the entire graph was transformed. Retyping of elements from the source model to different target types depending on further, context information was possible by using alternatives in our patterns. The first extension requiring real graph *rewriting* was solved easily with one additional declarative graph rewrite rule, in an intuitive syntax similar to the one specified by the authors. The second extension was not tackled directly due to the lack of a concrete syntax; but we presented an alternative solution (performing even better in a lot of cases) regarding the ultimate goal of a concrete syntax with our graph viewer yComp, delivering an excellent automatic layout of arbitrary data from a few lines of configuration information. The third extension was not tackled at all as we regard it off-topic at least for us.

References

[1] Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: Model Migration Case for TTC 2010. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/cases/ttc2010_submission_2_v2010-04-22.pdf (2010)

[2] Blomer, J., Geiß, R., Jakumeit, E.: The GrGen.NET User Manual. http://www.grgen.net (2010)

[3] Buchwald, S., Jakumeit, E., Kroll, M.: A GrGen.NET solution of the Program Comprehension case for the GraBaTs 2009 Contest (2009) http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/submissions/grabats2009_submission_13-final.pdf.

Migrating UML Activity Models with COPE

Markus Herrmannsdoerfer

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
herrmama@in.tum.de

Abstract. When a metamodel is adapted, the intention behind the metamodel adaptation is usually lost. To not lose the intention, COPE records the coupled evolution of metamodels and models, i. e. the model migration together with the metamodel adaptation. The coupled evolution is recorded as a sequence of coupled operations, each of which encapsulates a combination of metamodel adaptation and model migration. To further reduce the effort for building a model migration, recurring coupled operations can be reused. The recorded sequence of coupled operations can be used to migrate existing models. COPE is implemented based on the Eclipse Modeling Framework (EMF). In this paper, we apply COPE to migrate Activity models from UML 1.4 to 2.1.

1 COPE in a Nutshell

This paper explains the application of COPE to the model migration case [1] of the Transformation Tool Contest (TTC) 2010. As is depicted in Figure 1, COPE records the metamodel adaptation as a sequence of primitive changes in an explicit history model [2]. COPE supports two methods to form coupled operations, i.e. to attach a model migration to a sequence of primitive changes [3].

Reuse of recurring migration specifications allows to reduce the effort associated with building a model migration [4]. COPE thus provides *reusable coupled operations* which encapsulate metamodel adaptation and model migration in a metamodel-independent way. Reusable coupled operations are organized in a *library* which can be extended by declarations of new operations. The declaration is made independent of a specific metamodel through parameters, and may provide constraints to restrict the applicability of the operation. Listing 1 shows the interface and implementation of the operation Unfold Class which is required for the case study and whose semantics is informally explained in Section 4. Currently, COPE comes with over 60 reusable coupled operations. By means of studying real-life metamodel histories [3–5], we have shown that, in practice, most of the coupled evolution can be covered by reusable coupled operations. The case study also confirms these results, since only 1 of the 40 applied coupled operations is not reusable (see Table 1).

Specifications of model migration can become so specific to a certain metamodel that reuse makes no sense [4]. To be able to cover these specifications, COPE allows the user to manually define a *custom coupled operation*. In order

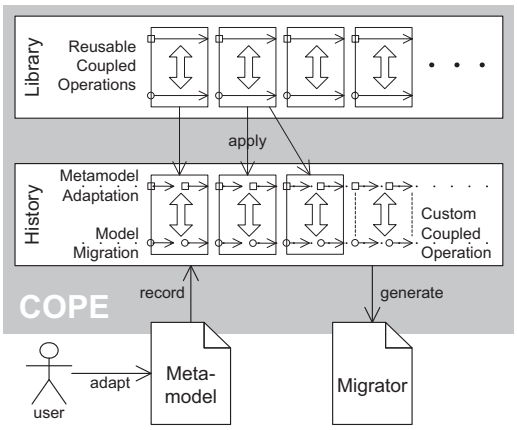


Fig. 1. Overview of COPE

to do so, the user has to manually encode a model migration for a recorded metamodel adaptation. To encode a model migration, COPE provides a language expressive enough to cater for complex model migrations. This language—which is also used to implement reusable coupled operations—is embedded into Groovy[1] to take advantage of its expressiveness. By softening the conformance of the model to the metamodel within a coupled operation, both metamodel adaptation and model migration can be specified as in-place transformation. A transaction mechanism ensures conformance at the boundaries of the coupled operation. Due to the in-place transformation, the language requires to specify only the difference for both metamodel and model. In the case study, only 1 coupled operation could not be covered by reusable coupled operations and thus required the implementation of a custom migration (see Table 1). This custom coupled operation is shown as part of Listing 2.

A *migrator* can be generated from the history model that allows for the batch migration of existing models. The migrator packages the sequence of coupled operations which can be executed to automatically migrate existing models. Listing 2 shows the generated migrator for the case study.

2 Forward Engineering the Coupled Evolution

To not lose the intention behind the metamodel adaptation, COPE is intended to be used already when adapting the metamodel. Therefore, COPE's user interface which is depicted in Figure 2 is directly integrated into the existing EMF *metamodel editor*. The user interface provides access to the *history model* in which COPE records the sequence of coupled operations. An initial history can be created for an existing metamodel by invoking *Create History* in the *operation browser* which also allows the user to *Release* the metamodel.

[1] `http://groovy.codehaus.org/`

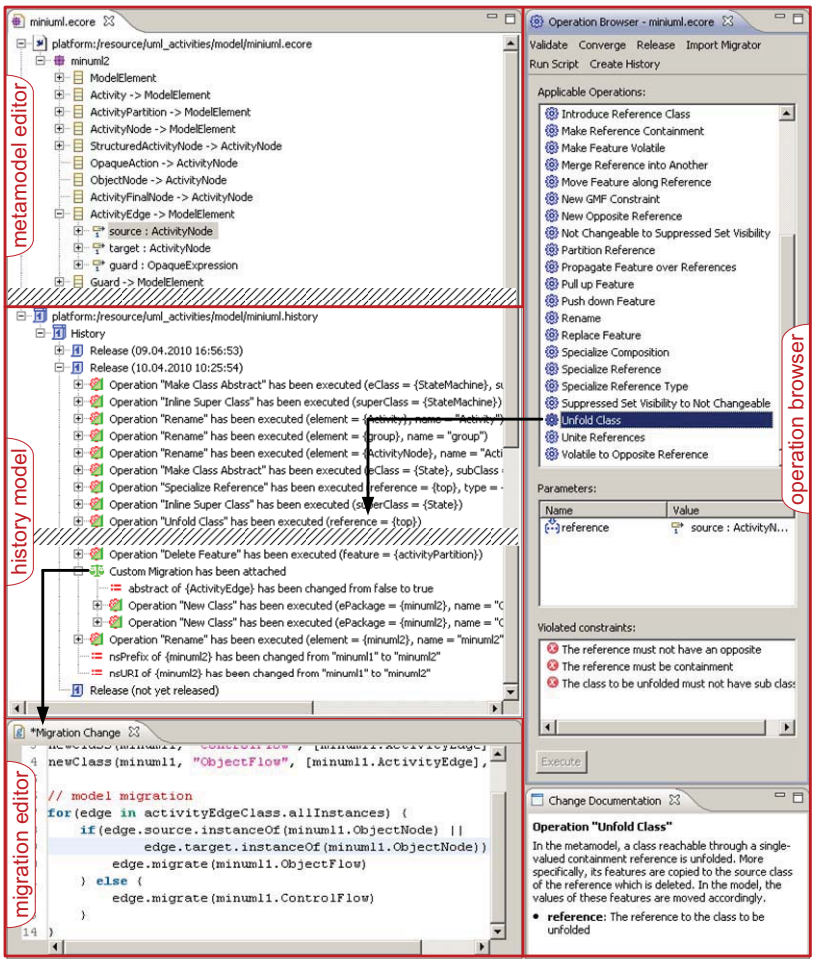


Fig. 2. User Interface of COPE (Shaded lines mean that elements have been cut out to fit the important information onto the paper)

The user can adapt the metamodel by applying reusable coupled operations through the *operation browser*. The operation browser allows to set the parameters of a reusable coupled operation, and gives feedback on its applicability based on the constraints. When a reusable coupled operation is executed, its application is automatically recorded in the history model. Figure 2 shows the operation Unfold Class being selected in the operation browser and recorded to the history model.

The user needs to perform a custom coupled operation only, in case no reusable coupled operation is available for the change at hand. First, the metamodel is directly adapted in the metamodel editor, in response to which the changes are automatically recorded in the history. A migration can later be at-

tached to the sequence of metamodel changes. Figure 2 shows the *migration editor* to encode the custom migration required for the case study.

3 Reverse Engineering the Coupled Evolution

If COPE was not used for adapting the metamodel, the intention behind the metamodel adaptation is already lost. In the case study, we only have access to the version of the metamodel before and after adaptation. Therefore, the history model needs to be recovered from the two metamodel versions.

COPE provides tool support to reverse engineer the history model. Figure 3 depicts the user interface to let a source metamodel version converge to a target metamodel version. As a prerequisite, the *source metamodel* version is loaded directly in the metamodel editor, together with its initial history. The *target metamodel* is displayed in a separate view which can be obtained by invoking *Converge* in the operation browser (see Figure 2) and selecting the file containing the target metamodel. This so-called *Convergence View* also displays the current *difference model* which results from the comparison between the source and target metamodel. The differences are linked to the metamodel elements from both source and target version to which they apply. Breaking changes [6] in the difference model—i. e. metamodel changes which necessitate a model migration—are highlighted in red. By means of the operation browser, the user can apply reusable coupled operations to bring the source metamodel nearer to the target metamodel. After an operation is executed on the source metamodel, the difference is automatically updated to reflect the changes. Changes in the difference model which are not breaking can be easily applied to the source metamodel by double-clicking on them.

4 Library of Reusable Coupled Operations

In this section, we list and explain all coupled operations that are needed for the case study. The coupled operations are classified according to their effect on existing models [7]. This allows to reason about the coupled evolution. Table 1 provides an overview over the coupled operations, their type, effect and number of occurrences.

4.1 Refactorings

Refactorings preserve the set of models that can be built. Consequently, the information contained in models is preserved in response to a refactoring.

Rename: A class or feature is renamed. Renaming is propagated to the model. In the case study, Rename is used to map the state diagram classes and features to activity diagram classes and features, respectively.

Inline Super Class: An abstract super class is inlined into its sub classes. On the metamodel level, the class is removed and its features are pushed down to the

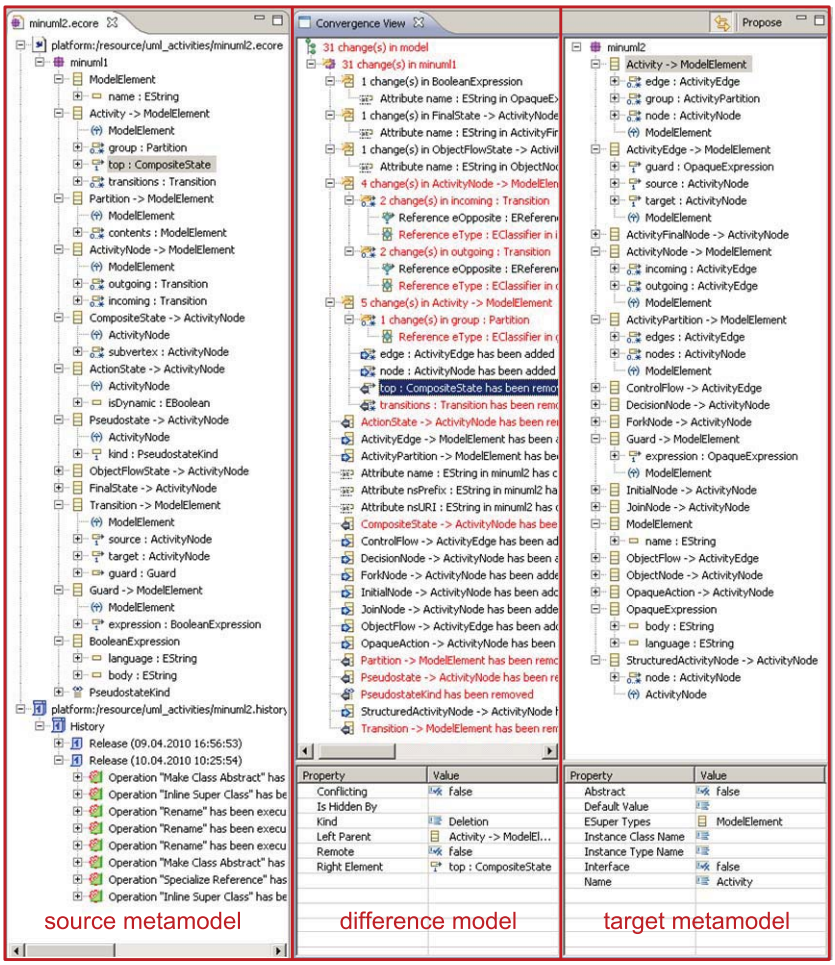


Fig. 3. Metamodel Convergence (Due to lack of space, the operation browser is not shown in this figure.)

sub classes. On the model level, the values of these features have to be moved accordingly. In the case study, Inline Super Class is used to remove classes for which there is no corresponding class in the new metamodel—e. g. *Statemachine* (line 3 in Listing 2) and *Pseudostate* (line 31).

Unfold Class: A single-valued containment reference is replaced by the features defined by its target class. On the metamodel level, the reference is deleted and the features of its target class are copied to its source class. On the model level, the values of these features are moved accordingly. Listing 1 shows the implementation of Unfold Class. In the case study, Unfold Class is e. g. used to replace the containment reference *top* in *Statemachine* with the features of the sub class *CompositeState* of its target class (line 10).

Table 1. Coupled Operations required for the case study

Coupled Operation	Type	Effect	#
Rename	Reusable	Refactoring	21
Inline Super Class	Reusable	Refactoring	3
Unfold Class	Reusable	Refactoring	2
Enumeration to Sub Classes	Reusable	Refactoring	1
Partition Reference	Reusable	Refactoring	1
Delete Feature	Reusable	Destructor	8
Make Class Abstract	Reusable	Destructor	2
Specialize Reference	Reusable	Destructor	1
Split Class	Custom	Constructor	1

Enumeration to Sub Classes: An enumeration attribute of a class is replaced by subclasses. On the metamodel level, the class is made abstract, and a subclass is created for each literal of the enumeration. The enumeration attribute is deleted and also the enumeration, if not used otherwise. On the model level, instances of the class are migrated to the appropriate subclass according to the value of the enumeration attribute. In the case study, Enumeration to Sub Classes is used to split *PseudoState* into a sub class for each literal of the enumeration *PseusoStateKind* (line 25).

Partition Reference: A reference is partitioned into a number of references according to its type. On the metamodel level, a sub reference is created for each subclass of the reference's type which needs to be abstract. Finally, the original reference is deleted. On the model level, the value of the reference is partitioned accordingly. In the case study, Partition Reference is used to partition the reference *contents* of *Partition* into references for each subclass of *ModelElement* (line 34).

4.2 Destructors

Destructors decrease the set of models that can be built. Consequently, information may be lost in models in response to a destructor.

Delete Feature: On the metamodel level, a feature is deleted. If the feature is a reference that has an opposite reference, then the opposite is also deleted. On the model level, the values of the features are deleted. In the case study, Delete Feature is used to delete features that are no longer available in the target metamodel, e.g. *isDynamic* of *ActionState*. Delete Feature is often applied to delete features that are created by earlier applications of other operations. For instance, the application of Unfold Class leads to more features than are actually part of the new metamodel (lines 11, 13 and 14).

Make Class Abstract: On the metamodel level, a class is made abstract. On the model level, instances of this class are migrated to a selected subclass. In the case study, classes are made abstract to be able to apply Inline Super Class. For instance, *State* is made abstract, and its instances are migrated to *ActionState* which is renamed to *OpaqueAction* (line 7).

Specialize Reference: A reference is specialized by restricting its type and/or multiplicity. On the model level, values no longer conforming to the new type or multiplicity are removed. In the case study, the type of *top* is specialized to *CompositeState* so that we are able to unfold its features into *StateMachine* (line 8).

4.3 Constructors

Constructors increase the set of models that can be built. Consequently, no information is lost in models due to a constructor, but new information can be added to models.

Split Class: A class is split into several sub classes. On the metamodel level, the class is made abstract and the sub classes are created. On the model level, instances of the class have to be migrated to one of the sub classes. Custom code is necessary to implement the logic to decide to which sub class a certain instance is migrated. In the case study, Split Class is used to split *Transition*—which is renamed to *ActivityEdge*—into *ControlFlow* and *ObjectFlow* (lines 41 to 56). If source or target of an *ActivityEdge* is an *ObjectNode*, then we migrate to *ObjectFlow*, otherwise to *ControlFlow*.

5 Discussion

The solution is discussed along the evaluation criteria of the case [1].

Correctness. The reverse engineered coupled evolution is correct in the sense that it produces the same model as the one provided with the case. Additionally, by classifying coupled operations, one can reason about their effect on existing models. To adapt the provided minimal original metamodel to the evolved metamodel, a number of coupled operations were necessary that may lead to information loss in existing models. This is however not the case for the provided original model, but may be the case for other models conforming to the original metamodel. To not lose information in these other models, additional metamodel elements of the complete metamodels may have to be added to the minimal metamodels.

Conciseness. We showed that most of the coupled evolution can be automated by reusable coupled operations, while only a very small amount (9 lines) of migration code needs to be handcoded. Therefore, the solution can be considered as very concise. However, to profit from reusable coupled operations, the user needs to have knowledge about the operations in the library. COPE supports the user to select an operation by only showing those operations in the operation browser that are applicable based on the currently selected element in the metamodel editor.

Clarity. For users familiar with typical model transformation languages, the coupled evolution may be difficult to understand. This is probably due to the fact that the model migration is modularized along the metamodel adaptation. However, the modularization allows to understand and reason about each coupled

operation separately. COPE provides additional tool support to interactively reconstruct the different metamodel versions, when browsing the metamodel history. To understand the model migration induced by the employed reusable coupled operations, knowledge about the library is helpful, though not necessary, as COPE provides a view that interactively shows the documentation for each reusable coupled operation.

6 Conclusion

In this paper, we presented the application of COPE to reverse engineer the coupled evolution of the provided case study. Even with the tool support for reverse engineering, effort was required to select the appropriate reusable coupled operations. To ease reverse engineering the coupled evolution, we envision a function that automatically proposes reusable coupled operations based on the difference between two metamodel versions. COPE is open source and can be obtained from its website[2]. The website also provides a screencast, documentation and several examples (including the case study from this paper). Moreover, COPE is about to be made available via the newly created Eclipse Project Edapt[3].

The presented case study is not a typical case study for model migration. In a typical metamodel adaptation scenario, only a rather small part of the metamodel is adapted. In model transformation languages which are not tailored for model migration, identity mappings need to be specified for metamodel elements that have not changed. Due to the minimality of the provided metamodel versions, nearly all the metamodel was adapted. Consequently, nearly no identity mappings are necessary in conventional model transformation languages which is rather untypical for model migration. Note that COPE does not require to specify identity mappings either in a typical metamodel adaptation scenario due to the in-place nature of the transformation.

Acknowledgments. This work was funded by the German Federal Ministry of Education and Research (BMBF), grants "SPES2020, 01IS08045A" and "Quamoco, 01IS08023B".

References

1. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: Model migration case for ttc 2010. In: TTC 2010: Proc. Transformation Tool Contest Workshop. (2010)
2. Herrmannsdoerfer, M.: Operation-based versioning of metamodels with cope. In: CVSM '09: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, Washington, DC, USA, IEEE Computer Society (2009) 49–54
3. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: ECOOP 2009 - Object-Oriented Programming. Volume 5653 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2009) 52–76

[2] `http://cope.in.tum.de`
[3] `http://www.eclipse.org/proposals/edapt/`

4. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: Model Driven Engineering Languages and Systems (MODELS 2008). Volume 5301/2008 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (October 2008) 645–659
5. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.: Language evolution in practice: The history of GMF. In: Software Language Engineering. Volume 5969 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2010) 3–22
6. Becker, S., Goldschmidt, T., Gruschko, B., Koziolek, H.: A process model and classification scheme for semi-automatic meta-model evolution. In: Proc. 1st Workshop MDD, SOA und IT-Management (MSI'07), GiTO-Verlag (2007) 35–46
7. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: ECOOP 2007 - Object-Oriented Programming. Volume 4609/2007 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2007) 600–624

A Extensions

The appendix discusses the extensions of the case [1]. The solution addresses both variants of the core migration task, but does not address the ArgoUML extensions.

A.1 Alternative Object Flow State Migration Semantics

COPE provides tool support to refactor the history model for the core model migration to integrate the semantic extension. Listing 3 shows the replacement for the custom coupled operation in Listing 2 (lines 41 to 56). First, a custom migration can be removed and another one can be attached. In the case study, we replaced the custom migration to split the class *ActivityEdge* by a custom migration to build instances for the subclass *ObjectFlow* (lines 2 to 19 in Listing 3). In this custom migration, edges incoming to and outgoing from *ObjectNode*s are replaced by direct *ObjectFlow*s. So this leaves us with two primitive changes (make class *ActivityEdge* abstract, create subclass *ControlFlow*) that have no longer a custom migration attached. Making the class *ActivityEdge* abstract is breaking and thus requires a migration. Second, metamodel changes can be reordered, as long as they respect dependencies between the changes. In the case study, we moved the changes to make class *ActivityEdge* abstract and to create its concrete subclass *ControlFlow* to after the new custom coupled operation. Third, primitive metamodel changes can be replaced by reusable coupled operations which perform the same change on the metamodel level. In the case study, we replaced the primitive change to make class *ActivityEdge* abstract with the reusable coupled operation Make Class Abstract. However, the semantics extension leads to more loss of information than the core migration, as instances of *ObjectNode* are deleted by the custom coupled operation.

A.2 XMI

COPE is not intended to bridge several technological spaces, but to provide support for incrementally adapting EMF metamodels. If we had a bridge that

generically migrates metamodels and models from XMI 1.2 to EMF, then COPE could be used on the result in a straightforward manner.

A.3 Concrete Syntax

The provided concrete syntax is also from a different technological space and hence its migration is currently not supported.

Listing 1. Reusable coupled operation Unfold Class

```
@description("In the metamodel, a class reachable through a single-valued " +
    "containment reference is unfolded. More specifically, its features are " +
    "copied to the source class of the reference which is deleted. In the "
    "model, the values of these features are moved accordingly.")
@label("Unfold Class")
unfoldClass = {
    @description("The reference to the class to be unfolded")
    EReference reference ->

    // variables
    def EClass unfoldedClass = reference.eReferenceType
    def EClass contextClass = reference.eContainingClass
    def List<EStructuralFeature> features =
        new ArrayList(unfoldedClass.eAllStructuralFeatures)

    // constraints
    assert reference.eOpposite == null :
        "The reference must not have an opposite"
    assert reference.upperBound == 1 :
        "The multiplicity of the reference must be single-valued"
    assert reference.containment :
        "The reference must be containment"
    assert unfoldedClass.eSubTypes.isEmpty() :
        "The class to be unfolded must not have sub classes"

    // metamodel adaptation
    def unfoldedFeatures = []
    for(feature in features) {
        def unfoldedFeature = feature.clone()
        unfoldedFeatures.add(unfoldedFeature)
        // ensure uniqueness of the name of the unfolded feature
        if(contextClass.getEStructuralFeature(feature.name) != null) {
            unfoldedFeature.name = unfoldedFeature.name + "_" + unfoldedClass.name
        }
        contextClass.eStructuralFeatures.add(unfoldedFeature)
        if(feature instanceof EReference && feature.eOpposite != null) {
            def foldedOpposite = feature.eOpposite.clone()
            foldedOpposite.eType = contextClass
            // ensure uniqueness of the name of the opposite feature
            foldedOpposite.name = foldedOpposite.name + "_" + contextClass.name
            feature.eType.eStructuralFeatures.add(foldedOpposite)
            foldedOpposite.eOpposite = unfoldedFeature
        }
    }
    reference.delete()

    // model migration
    for(contextElement in contextClass.allInstances) {
        unfoldedElement = contextElement.unset(reference)
        if(unfoldedElement != null) {
            int i = 0;
            for(feature in features) {
                contextElement.set(unfoldedFeatures[i],
                    unfoldedElement.unset(feature))
                i++;
            }
            unfoldedElement.delete()
        }
    }
}
```

Listing 2. Migrator code generated from the reverse engineered history model

```
// reusable coupled operations
makeAbstract(minuml1.StateMachine, minuml1.ActivityGraph)
inlineSuperClass(minuml1.StateMachine)
rename(minuml1.ActivityGraph, "Activity")
rename(minuml1.Activity.partition, "group")
rename(minuml1.StateVertex, "ActivityNode")
makeAbstract(minuml1.State, minuml1.ActionState)
specializeReference(minuml1.Activity.top, minuml1.CompositeState, 1, 1)
inlineSuperClass(minuml1.State)
unfoldClass(minuml1.Activity.top)
deleteFeature(minuml1.Activity.name_CompositeState)
rename(minuml1.Activity.transitions, "edge")
deleteFeature(minuml1.Activity.incoming)
deleteFeature(minuml1.Activity.outgoing)
rename(minuml1.Activity.subvertex, "node")
rename(minuml1.Partition, "ActivityPartition")
rename(minuml1.CompositeState, "StructuredActivityNode")
rename(minuml1.StructuredActivityNode.subvertex, "node")
rename(minuml1.Transition, "ActivityEdge")
unfoldClass(minuml1.ActivityEdge.guard)
deleteFeature(minuml1.ActivityEdge.name_Guard)
rename(minuml1.BooleanExpression, "OpaqueExpression")
rename(minuml1.ActionState, "OpaqueAction")
deleteFeature(minuml1.OpaqueAction.isDynamic)
enumerationToSubClasses(minuml1.Pseudostate.kind, minuml1)
rename(minuml1.initial, "InitialNode")
rename(minuml1.join, "JoinNode")
rename(minuml1.fork, "ForkNode")
rename(minuml1.junction, "DecisionNode")
rename(minuml1.ActivityEdge.expression, "guard")
inlineSuperClass(minuml1.Pseudostate)
rename(minuml1.ObjectFlowState, "ObjectNode")
rename(minuml1.FinalState, "ActivityFinalNode")
partitionReference(minuml1.ActivityPartition.contents)
rename(minuml1.ActivityPartition.activityEdge, "edges")
rename(minuml1.ActivityPartition.activityNode, "nodes")
deleteFeature(minuml1.ActivityPartition.guard)
deleteFeature(minuml1.ActivityPartition.activity)
deleteFeature(minuml1.ActivityPartition.activityPartition)

// custom coupled operation: Split Class
// metamodel adaptation
activityEdgeClass = minuml1.ActivityEdge
activityEdgeClass.'abstract' = true
newClass(minuml1, "ControlFlow", [minuml1.ActivityEdge], false)
newClass(minuml1, "ObjectFlow", [minuml1.ActivityEdge], false)

// model migration
for(edge in activityEdgeClass.allInstances) {
    if(edge.source.instanceOf(minuml1.ObjectNode) ||
            edge.target.instanceOf(minuml1.ObjectNode)) {
        edge.migrate(minuml1.ObjectFlow)
    } else {
        edge.migrate(minuml1.ControlFlow)
    }
}

// reusable coupled operations
rename(minuml1Package, "minuml2")
minuml1Package = minuml1
minuml1Package.nsPrefix = "minuml2"
minuml1Package.nsURI = "minuml2"
```

Listing 3. Update for the extension of the migration semantics

```
...
// custom coupled operation
// metamodel adaptation
newClass(minuml1, "ObjectFlow", [minuml1.ActivityEdge], false)

//model migration
for(on in minuml1.ObjectNode.allInstances) {
    for(i in on.incoming) {
        for(o in on.outgoing) {
            def of = minuml1.ObjectFlow.newInstance()
            on.container.edge.add(of)
            of.source = i.source
            of.target = o.target
        }
    }
    for(i in on.incoming) i.delete()
    for(o in on.outgoing) o.delete()
    on.delete()
}

// reusable coupled operations
newClass(minuml1, "ControlFlow", [minuml1.ActivityEdge], false)
makeAbstract(minuml1.ActivityEdge, minuml1.ControlFlow)
...
```

UML Model Migration with PETE

Bernhard Schätz

fortiss GmbH
Guerickestr. 25, 80805 München, Germany
schaetz@fortiss.org

Abstract. With models becoming a common-place in software and systems development, the support of automatic transformations of those models is an important asset to increase the efficiency and improve the quality of the development process. However, the definition of transformations still is quite complex. Several approaches – from more imperative to more declarative styles – have been introduced to support the definition of such transformations. Here, we show how a *completely* declarative relational style based on the interpretation of a model as single structured term can be used to provide a transformation mechanism allowing a simple, precise, and modular specification of transformations for the EMF Ecore platform, using a Prolog rule-based mechanism. The approach is demonstrated ion tex context of migrating Activity models from UML 1.4 to 2.1.

1 PETE in a Nutshell

To construct formalized descriptions of a system under development, a 'syntactic vocabulary' is needed. This conceptual model[1] characterizes all possible system models built from the *modeling concepts and their relations* used to construct a description of a system; typically, class diagrams are used to describe them.

1.1 Structure of the Model

The transformation framework provides mechanisms for a pure (i.e., side-effect free) declarative, rule-based approach to model transformation, accessing EMF Ecore-based models [SBPM07]. Based on the conceptual model, a system model consists of sets of elements (each described as a conceptual entity and its attribute values) and relations (each described as a pair of conceptual entities), syntactically represented as a Prolog term. Since these elements and relations are instances of classes and associations taken from an EMF Ecore model, the structure of the Prolog term – representing an instance model – is inferred from the structure of that model. The structure of the model is built using only simple

[1] In the context of technologies like the Meta Object Facility, the class diagram-like definition of a conceptual model is generally called *meta model*.

ModelTerm	::=	*PackageTerm*
PackageTerm	::=	*Functor*(*PackagesTerm*,*ClassesTerm*,*AssociationsTerm*)
PackagesTerm	::=	[] | [*PackageTerm* (,*PackageTerm*)*]
ClassesTerm	::=	[] | [*ClassTerm* (,*ClassTerm*)*]
ClassTerm	::=	*Functor*(*ElementsTerm*)
ElementsTerm	::=	[] | [*ElementTerm* (,*ElementTerm*)*]
ElementTerm	::=	*Functor*(*Entity*(,*AttributeValue*)*)
Entity	::=	*Atom*
AttributeValue	::=	*Atom*
AssociationsTerm	::=	[] | [*AssociationTerm*(,*AssociationTerm*)*]
AssociationTerm	::=	*Functor*(*RelationsTerm*)
RelationsTerm	::=	[] | [*RelationTerm*(,*RelationTerm*)*]
RelationTerm	::=	*Functor*(*Entity*,*Entity*)

Table 1. The Prolog Structure of a Model Term

elementary Prolog constructs, namely compound functor terms and list terms. To access a model, the framework provides predicates to deconstruct and reconstruct a term representing a model. [Sch08] describes the model in more detail.

A *model term* describes an instance of a EMF Ecore model. Each model term is a list of package terms, one for each packages of the EMF Ecore model. Each *package term*, in turn, describes the content of the package instance. It consists of a functor, identifying the package, with a sub-packages term, a classes terms, and an associations term as its argument. The sub-packages term describes the sub-packages of the package; it is a list of package terms.

The *classes term* describes the EClasses of the corresponding package. It is a list of class terms, one for each EClass. Each *class term* consists of a functor, identifying the class, and an elements term. An *elements term* describes the collection of objects instantiating this class, and thus is a list of element terms. Finally, an *element term* consists of a functor, identifying the class this object belongs to, with an entity identifying the element and attributes as arguments. Each of the attributes are atomic representations of the corresponding values of the attributes of the represented object. The entity is a regular atom, unique for each element term.

Similarly to an elements term, each *associations term* describes the associations, i.e., the instances of the EReferences of the EClasses, for the corresponding package. Again, it is a list of association terms, with each *association term* consisting of a functor, identifying the association, and an relations term, describing the content of the association. The *relations term* is a list of relation terms, each *relation term* consisting of a functor, identifying the relation, and the entity identificators of the related objects. In detail, the Prolog model term has the structure shown in Table 1 in the BNF notation with corresponding *non-terminals* and *terminals*.[2]

[2] While actually a *ModelTerm* consists of a set of *PackageTerms*, here for simplification purposes only one *PackageTerm* is assumed.

The functors of the compound terms are deduced from the EMF Ecore model: The functor of a PackageTerm from the name of the EPackage; the functor of a ClassTerm from the name of the EClass; the functor of an AssociationTerm from the name of the EReference. Similarly, the atoms of the attributes are deduced from the instance of the EMF Ecore model, which the model term is representing: The entity atom corresponds to the object identificator of an instance of a EClass, while the attribute corresponds to the attribute value of an instance of an EClass.

1.2 Construction Predicates

In a strictly declarative rule-based approach, the transformation is described in terms of a predicate, relating the models before and after the transformation. Therefore, mechanisms are needed in form of predicates to deconstruct a model into its parts as well as to construct a model from its parts. As the structure of the model is defined using only compound functor terms and list terms, only two forms of predicates are needed: union and composition operations.

List Construction The(de)construction of lists is managed by means of the union predicate `union/3` with template[3] `union(?Left,?Right,?All)` such that `union(Left,Right,All)` is true if all elements of list `All` are either elements of `Left` or `Right`, and vice versa.

Compound Construction Since the compound structures used to build the model instances depend on the actual structure of the EMF Ecore model, only the general schemata used are described. In all three schemata – package, class/element, or association/relation – the name of the package, class, or relation is used as the name of the predicate for the compound construction.

Packages For (de)construction of packages, package predicates of the form `package/4` are used with template `package(?Package,?Subpackages, ?Classes,?Associations)` where package is the name of the package (de)constructed.

Classes and Elements For (de)construction of – non-abstract – classes/elements, class/element predicates of the form `class/2` and `class/N+2` are used where N is the number of the attributes of the corresponding class, with templates `class(?Class, ?Elements)` and `class(?Element,?Entity,?Attr1,...,?AttrN)` where class is the name of the class and element (de)constructed. The class predicate is true if `Class` is the list of `Objects`; it is used to deconstruct a class into its list of objects, and vice versa. Similarly, the element predicate is true if `Element` is an `Entity` with attributes `Attr1`,...,`AttrN`; it can be used to deconstruct an element into

[3] According to standard convention, arbitrary/input/output arguments of predicates are indicated by `?`/`+`/`-`.

its entity and attributes, to construct an element from an entity and attributes (e.g. to change the attributes of an element), or to construct a new element including its entity from the attributes.

Association and Relation Compounds For (de)construction of associations and relations, association and relation predicate of the form `association/2` and `association/3` are used with templates `association(?Association,?Relations)` and `association(?Relation,?Entity1,?Entity2)` where association is the name of the association and relation constructed/deconstructed. The relation predicate is true if `Association` is the list of `Relations`; it is generally used to deconstruct an association into its list of relations, and vice versa. Similarly, the relation predicate is true if `Relation` associates `Entity1` and `Entity2`; it is used to deconstruct a relation into its associated entities and vice versa.

2 Transformation Definition

The conceptual model and its structure defined in Section 1 was introduced to define transformations of system models. In a relational approach to model transformations, a transformation – like the migrating from UML1.4 Activity Diagrams to UML 2.1 Activity Diagrams – is described as a relation between the model prior to the transformation (e.g., the UML 1.4 Diagram) and the model after the transformation (e.g., the UML 2.1 Activity Diagram). In this section, the basic principles of describing transformations as relations are described.

2.1 Transformations as Relations

In case of the migration operation, the relation describing the transformation has the interface `migrate(UML1,UML2)` with parameter `UML1` for the UML1.4 model and parameter `UML2` for the UML 2.1 model. In the relational approach presented here, a transformation is basically described by breaking down the pre-model into its constituents and build up the post-model from those constituents using the relations from Section 1, potentially adding or removing elements and relations. With `UML1` taken from the conceptual domain for UML1 Activity diagrams and packaged in a single package *minuml1* with no sub-packages, it can be decomposed in contained classes (e.g., *PseudoState*) and associations (e.g., *partition*) as shown in Section B.2 Note that the relation is bidirectional: Besides migrating a UML 1.4 model into a UML 2.1 model, it can also be used to migrate back from the UML 2.1 to the UML 1.4 model.

Besides using the basic relations to construct and deconstruct models (and add or remove elements and relations, as shown in the next subsection), new relations can be defined to support a modular description of transformation, decomposing rules into sub-rules. E.g., in the `migrate` relation, the transformation can be decomposed into the migration of classes and associations; for the latter, then, e.g., a sub-relation `migratePseudoState` with corresponding rules is introduced, as shown in Subsection A.2.

2.2 Transformations as Rules

To define the transformation steps for migrating classes and associations, relations like `migrateActionState(UML1ActionStates,UML2OpaqueActions)` or `migratepartition(UML1partition,UML2group)` are used. To define these rules as shown in Sections A.2 and A.3, the conceptual models of the source and the target models and their structured representation introduced in Section 1 are used. Note that this rule-based description allows to compose complex transformations by simple application of rules in the body of another rule (like `migrateActionState` in `migrate`).

3 Conclusion and Outlook

The PETE transformation framework – provided as an Eclipse PlugIn [Sch09] – supports the transformation of EMF Ecore models using a declarative relational style and allows a simple, precise, and modular specification of transformation relations on the problem- rather than the implementation-level. By including operational aspects, this form of specification can be tuned to ensure an efficient execution. The migration case study has shown that the use of purely declarative relational transformations is applicable to practical problems. Less than 150 LOCs are necessary to specify the migration. Even though in general a declarative specification is less efficient than a imperative form, the approach is feasible for medium-sized problems (i.e., up to several 10K of modeling elements).

Some specific advantages visible in the case study concern the direct expressiveness of the transformation language. E.g., the migration does not involve the construction of correspondence graph to control the migration. Furthermore, rules in general follow a very simple recursion scheme, possible involving structural rules for distinction of structural cases. Even complex tasks – like the migration of ObjectFlowStates to ObjectFlows and the corresponding elimination of Transitions – can be managed with these scheme. By implementing higher-order predicates for these schemes – like the `renameElements` and `renameRelations` predicates shown in B – migration operators for these standard tasks can be provided, drastically simplifying the migration complexity. To support the effective development, furthermore debugging can actually performed on the level of the specification language.

Further advantages of a declarative relational approach not applied here include the bidirectional transformations. Furthermore, this style of transformations is helpful for the formal verification of correctness conditions of these transformations. Using tools like *Isabelle/HOL*, the verification process can be automized to a large extent. Finally, the declarative relational style can also be used to support a search-based design-space exploration involving backtracking, e.g., in the case of multiple migration solutions for a specific model. As the major drawback of the approach, PETE is not specifically targeting model migration or metamodel/mode co-evolution, as e.g., tools like COPE. Therefore, it requires to handcraft the transformations and does not support the generation of the transformation rules.

References

[SBPM07] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison Wesley Professional, 2007. Second Edition.

[Sch08] Bernhard Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. In Eric Van Wyk Dragan Gasevic, Ralf Laemmel, editor, *Software Language Engineering*, LNCS. Springer, 2008.

[Sch09] Bernhard Schätz. Prolog EMF Transformation Eclipse-PlugIn. `www4.in.tum.de/~schaetz/PETE`, 2009.

A PETE Code Listing

A.1 (De-)Constructing The UML Models

```
1  migrate(UML1,UML2) :-
2      minuml1(UML1Pack,[],UML1Class,UML1Assoc), union([UML1Pack],[],UML1),
3      Pseudostate(UML1Pseudostate,UML1Pseudostates), FinalState(UML1FinalState,
           UML1FinalStates),
4      ActionState(UML1ActionState,UML1ActionStates), Transition(UML1Transition,
           UML1Transitions),
5      Guard(UML1Guard,UML1Guards), BooleanExpression(UML1Expression,
           UML1Expressions),
6      ObjectFlowState(UML1ObjectFlowState,UML1ObjectFlowStates), Partition(
           UML1Paritition,UML1Partitions),
7      ActivityGraph(UML1ActivityGraph,UML1ActivityGraphs),
8      union([UML1Pseudostate,UML1FinalState,UML1ActionState,UML1Transition,
           UML1Guard,UML1Expression,UML1ObjectFlowState,UML1Paritition,
           UML1ActivityGraph],RestUML1Class,UML1Class),
9      incoming(UML1incoming,UML1incomings), outgoing(UML1outgoing,
           UML1outgoings),
10     expression(UML1expression,UML1expressions), subvertex(UML1subvertex,
           UML1subvertices),
11     contents(UML1content,UML1contents), transitions(UML1transition,
           UML1transitions),
12     guard(UML1guard,UML1guards), type(UML1type,UML1types),
13     partition(UML1partition,UML1partitions),
14     union([UML1incoming,UML1outgoing,UML1expression,UML1subvertex,
           UML1content,UML1transition,UML1guard,UML1type,UML1partition],
           RestUML1Assoc,UML1Assoc),
15     migrateActivityGraph(UML1ActivityGraphs,Activity,UML2Activities),
16     migratePseudostate(UML1Pseudostates,UML2InitialNodes,UML2ForkNodes,
           UML2JoinNodes,UML2DecisionNodes),
17     migrateFinalState(UML1FinalStates,UML2ActivityFinalNodes),
18     migrateActionState(UML1ActionStates,UML2OpaqueActions),
19     migrateTransition(UML1Transitions,UML2ActivityEdges),
```

```
20    migrateGuard(UML1Guards,UML1Expressions,UML1expressions,
          UML2OpaqueExpressions),
21    migrateActivityEdge(UML1ObjectFlowStates,UML2ActivityEdges,UML1incomings
          ,UML1outgoings,UML2incomings,UML2outgoings,UML2ObjectFlows,
          UML2ControlFlows),
22    migratePartition(UML1Partitions,UML2ActivityPartitions),
23    migratesubvertex(UML1subvertices,UML1contents,Activity,UML2ObjectFlows,
          UML2nodes,UML2Partnodes,UML2subvertexEdges,UML2Partedges),
24    migratetransitions(UML1transitions,UML2ControlFlows,UML2transitionEdges),
25    migratepartition(UML1partitions,UML2groups),
26    Activity(UML2Activity,UML2Activities), ObjectFlow(UML2ObjectFlow,
          UML2ObjectFlows), ControlFlow(UML2ControlFlow,UML2ControlFlows),
          InitialNode(UML2InitialNode,UML2InitialNodes),
27    ForkNode(UML2ForkNode,UML2ForkNodes), JoinNode(UML2JoinNode,
          UML2JoinNodes), DecisionNode(UML2DecisionNode,UML2DecisionNodes),
28    ActivityFinalNode(UML2FinalNode,UML2ActivityFinalNodes), ActivityPartition(
          UML2Partition,UML2ActivityPartitions),
29    OpaqueExpression(UML2OpaqueExpression,UML2OpaqueExpressions),
          OpaqueAction(UML2OpaqueAction,UML2OpaqueActions),
30    union([UML2Activity,UML2ObjectFlow,UML2ControlFlow,UML2InitialNode,
          UML2ForkNode,UML2JoinNode,UML2DecisionNode,UML2FinalNode,
          UML2Partition,UML2OpaqueExpression,UML2OpaqueAction],[],UML2Class),
31    node(UML2node,UML2nodes), union(UML2subvertexEdges,UML2transitionEdges,
          UML2edges), edge(UML2edge,UML2edges), group(UML2group,UML2groups),
          nodes(UML2Partnode,UML2Partnodes),
32    edges(UML2Partedge,UML2Partedges), outgoing(UML2outgoing,UML2outgoings),
          incoming(UML2incoming,UML2incomings),
33    guard(UML2guard,UML1guards), type(UML2type,UML1types),
34    union([UML2node,UML2edge,UML2group,UML2Partnode,UML2Partedge,
          UML2outgoing,UML2incoming,UML2guard,UML2type],[],UML2Assoc),
35    minuml2(UML2Pack,[],UML2Class,UML2Assoc), union([UML2Pack],[],UML2).
```

A.2 Migrating Classes

```
1  migrateActivityGraph(UML1ActivityGraphs,Elem,UML2Activities) :-
2     ActivityGraph(ActivityGraph,Elem,Name), union([ActivityGraph],[],
          UML1ActivityGraphs),
3     Activity(Activity,Elem,Name), union([Activity],[],UML2Activities).
4
5  migratePseudostate(UML1Pseudostates,UML2InitialNodes,UML2ForkNodes,
       UML2JoinNodes,UML2DecisionNodes) :-
6     Pseudostate(Pseudostate,Elem,Name,initial), union([Pseudostate],RestStates,
          UML1Pseudostates),
7     migratePseudostate(RestStates,RestNodes,UML2ForkNodes,UML2JoinNodes,
          UML2DecisionNodes),
8     InitialNode(InitialNode,Elem,Name), union([InitialNode],RestNodes,
          UML2InitialNodes).
9  migratePseudostate(UML1Pseudostates,UML2InitialNodes,UML2ForkNodes,
       UML2JoinNodes,UML2DecisionNodes) :-
```

```
10      Pseudostate(Pseudostate,Elem,Name,fork), union([Pseudostate],RestStates,
            UML1Pseudostates),
11      migratePseudostate(RestStates,UML2InitialNodes,RestNodes,UML2JoinNodes,
            UML2DecisionNodes),
12      ForkNode(ForkNode,Elem,Name), union([ForkNode],RestNodes,UML2ForkNodes).
13  migratePseudostate(UML1Pseudostates,UML2InitialNodes,UML2ForkNodes,
        UML2JoinNodes,UML2DecisionNodes) :-
14      Pseudostate(Pseudostate,Elem,Name,join), union([Pseudostate],RestStates,
            UML1Pseudostates),
15      migratePseudostate(RestStates,UML2InitialNodes,UML2ForkNodes,RestNodes,
            UML2DecisionNodes),
16      JoinNode(JoinNode,Elem,Name), union([JoinNode],RestNodes,UML2JoinNodes).
17  migratePseudostate(UML1Pseudostates,UML2InitialNodes,UML2ForkNodes,
        UML2JoinNodes,UML2DecisionNodes) :-
18      Pseudostate(Pseudostate,Elem,Name,junction), union([Pseudostate],RestStates,
            UML1Pseudostates),
19      migratePseudostate(RestStates,UML2InitialNodes,UML2ForkNodes,
            UML2JoinNodes,RestNodes),
20      DecisionNode(DecisionNode,Elem,Name), union([DecisionNode],RestNodes,
            UML2DecisionNodes).
21  migratePseudostate([],[],[],[],[]).
22
23
24  migrateFinalState(UML1FinalStates,UML2ActivityFinalNodes) :-
25      FinalState(FinalState,Elem,Name), union([FinalState],RestStates,
            UML1FinalStates),
26      migrateFinalState(RestStates,RestNodes),
27      ActivityFinalNode(ActivityFinalNode,Elem,Name), union([ActivityFinalNode],
            RestNodes,UML2ActivityFinalNodes).
28  migrateFinalState([],[]).
29
30  migrateActionState(UML1ActionStates,UML2OpaqueActions) :-
31      ActionState(ActionState,Elem,Name,Dynamics), union([ActionState],RestStates,
            UML1ActionStates),
32      migrateActionState(RestStates,RestNodes),
33      OpaqueAction(OpaqueAction,Elem,Name), union([OpaqueAction],RestNodes,
            UML2OpaqueActions).
34  migrateActionState([],[]).
35
36  migrateTransition(UML1Transitions,UML2ActivityEdges) :-
37      Transition(Transition,Elem,Name), union([Transition],RestTransitions,
            UML1Transitions),
38      migrateTransition(RestTransitions,RestEdges),
39      ActivityEdge(ActivityEdge,Elem,Name), union([ActivityEdge],RestEdges,
            UML2ActivityEdges).
40  migrateTransition([],[]).
41
42
43  migrateGuard(UML1Guards,UML1Expressions,UML1expressions,
        UML2OpaqueExpressions) :-
```

```
44      Guard(Guard,GuardElem,Name), union([Guard],RestGuards,UML1Guards),
45      BooleanExpression(BooleanExpression,ExpressionElem,Language,Body), union([
            BooleanExpression],RestBooleanExpressions,UML1Expressions),
46      expression(ExpressionRel,GuardElem,ExpressionElem), union([ExpressionRel],
            RestExpressions,UML1expressions),
47      migrateGuard(RestGuards,RestBooleanExpressions,RestExpressions,
            RestOpaqueExpressions),
48      OpaqueExpression(OpaqueExpression,GuardElem,Language,Body), union([
            OpaqueExpression],RestOpaqueExpressions,UML2OpaqueExpressions).
49  migrateGuard([],[],[],[]).
50
51  migrateActivityEdge(UML1ObjectFlowStates,UML2ActivityEdges,UML1incoming,
        UML1outgoing,UML2incoming,UML2outgoing,UML2ObjectFlows,
        UML2ControlFlows) :-
52      ObjectFlowState(ObjectFlowState,StateElem,Name), union([ObjectFlowState],
            RestObjectFlowStates,UML1ObjectFlowStates),
53      incoming(IncomingFlowState,StateElem,Incoming), union([IncomingFlowState],
            Restincoming,UML1incoming),
54      outgoing(FlowStateOutgoing,StateElem,Outgoing), union([FlowStateOutgoing],
            Restoutgoing,UML1outgoing),
55      incoming(OutgoingTarget,Target,Outgoing), union([OutgoingTarget],
            RestUML1incoming,Restincoming),
56      outgoing(SourceIncoming,Source,Incoming), union([SourceIncoming],
            RestUML1outgoing,Restoutgoing),
57      ActivityEdge(IncomingEdge,Incoming,IncomingName), ActivityEdge(OutgoingEdge
            ,Outgoing,OutgoingName),
58      union([IncomingEdge,OutgoingEdge],RestActivityEdges,UML2ActivityEdges),
59      migrateActivityEdge(RestObjectFlowStates,RestActivityEdges,RestUML1incoming,
            RestUML1outgoing,RestUML2incoming,RestUML2outgoing,
            UML2RestObjectFlows,UML2ControlFlows),
60      ObjectFlow(ObjectFlow,StateElem,Name), union([ObjectFlow],
            UML2RestObjectFlows,UML2ObjectFlows),
61      outgoing(SourceStateElem,Source,StateElem), union([SourceStateElem],
            RestUML2outgoing,UML2outgoing),
62      incoming(StateElemTarget,Target,StateElem), union([StateElemTarget],
            RestUML2incoming,UML2incoming).
63  migrateActivityEdge([],UML2ActivityEdges,UML1incoming,UML1outgoing,
        UML2incoming,UML2outgoing,[],UML2ControlFlows) :-
64      ActivityEdge(ActivityEdge,Elem,Name), union([ActivityEdge],RestActivityEdges,
            UML2ActivityEdges),
65      migrateActivityEdge([],RestActivityEdges,UML1incoming,UML1outgoing,
            UML2incoming,UML2outgoing,[],RestControlFlows),
66      ControlFlow(ControlFlow,Elem,Name), union([ControlFlow],RestControlFlows,
            UML2ControlFlows).
67  migrateActivityEdge([],[],Incoming,Outgoing,Incoming,Outgoing,[],[]).
68
69  migratePartition(UML1Partitions,UML2ActivityPartitions) :-
70      Partition(Partition,Elem,Name), union([Partition],RestPartitions,UML1Partitions
            ),
71      migratePartition(RestPartitions,RestActivityPartitions),
```

```
72      ActivityPartition(ActivityPartition,Elem,Name), union([ActivityPartition],
            RestActivityPartitions,UML2ActivityPartitions).
73  migratePartition([],[]).
```

A.3 Migrating Associations

```
1   migratesubvertex(UML1subvertex,UML1contents,Activity,UML2ObjectFlows,
        UML2node,UML2nodes,UML2edge,UML2edges) :-
2       subvertex(Subvertex,State,Vertex), union([Subvertex],RestSubvertex,UML1subvertex
            ),
3       ObjectFlow(StateVertex,Vertex,Name), union([StateVertex],RestObjectFlows,
            UML2ObjectFlows),
4       contents(Content,Partition,Vertex), union([Content],RestContents,UML1contents),
5       migratesubvertex(RestSubvertex,RestContents,Activity,RestObjectFlows,UML2node
            ,UML2nodes,UML2edge,RestEdges),
6       edges(Edges,Partition,Vertex), union([Edges],RestEdges,UML2edges).
7   migratesubvertex(UML1subvertex,UML1contents,Activity,UML2ObjectFlows,
        UML2node,UML2nodes,UML2edge,UML2edges) :-
8       subvertex(Subvertex,State,Vertex), union([Subvertex],RestSubvertex,UML1subvertex
            ),
9       contents(Content,Partition,Vertex), union([Content],RestContents,UML1contents),
10      migratesubvertex(RestSubvertex,RestContents,Activity,UML2ObjectFlows,
            UML2node,RestNodes,UML2edge,UML2edges),
11      nodes(Nodes,Partition,Vertex), union([Nodes],RestNodes,UML2nodes).
12  migratesubvertex(UML1subvertex,[],Activity,UML2ObjectFlows,UML2node,
        UML2nodes,UML2edge,UML2edges) :-
13      subvertex(Subvertex,State,Vertex), union([Subvertex],RestSubvertex,UML1subvertex
            ),
14      ObjectFlow(StateVertex,Vertex,Name), union([StateVertex],RestObjectFlows,
            UML2ObjectFlows),
15      migratesubvertex(RestSubvertex,[],Activity,RestObjectFlows,UML2node,
            UML2nodes,RestEdge,UML2edges),
16      edge(Edge,Activity,Vertex), union([Edge],RestEdge,UML2edge).
17  migratesubvertex(UML1subvertex,[],Activity,UML2ObjectFlows,UML2node,
        UML2nodes,UML2edge,UML2edges) :-
18      subvertex(Subvertex,State,Vertex), union([Subvertex],RestSubvertex,UML1subvertex
            ),
19      migratesubvertex(RestSubvertex,[],Activity,UML2ObjectFlows,RestNode,
            UML2nodes,UML2edge,UML2edges),
20      node(Node,Activity,Vertex), union([Node],RestNode,UML2node).
21  migratesubvertex([],[],Activity,UML2ObjectFlows,[],[],[],[]).
22
23  migratetransitions(UML1transitions,UML2ControlFlows,UML2edge) :-
24      transitions(TransitionAssoc,StateMachine,Transition), union([TransitionAssoc],
            RestTransitions,UML1transitions),
25      ControlFlow(ControlFlow,Transition,Name), union([ControlFlow],
            RestControlFlows,UML2ControlFlows),
26      migratetransitions(RestTransitions,RestControlFlows,RestEdges),
27      edge(EdgeAssoc,StateMachine,Transition), union([EdgeAssoc],RestEdges,
            UML2edge).
```

```
28 migratetransitions(UML1transitions,UML2ControlFlows,UML2edge) :-
29     transitions(TransitionAssoc,StateMachine,Transition), union([TransitionAssoc],
           RestTransitions,UML1transitions),
30     migratetransitions(RestTransitions,UML2ControlFlows,UML2edge).
31 migratetransitions([],UML2ControlFlows,[]).;
32
33 migratepartition(UML1partition,UML2group) :-
34     partition(PartitionAssoc,StateVertex,Partition), union([PartitionAssoc],
           RestPartitions,UML1partition),
35     migratepartition(RestPartitions,RestGroup),
36     group(GroupAssoc,StateVertex,Partition), union([GroupAssoc],RestGroup,
           UML2group).
37 migratepartition([],[]).
```

B PETE Code Listing - Higher Order Predicates

B.1 Using Higher-Order Predicates: Rename

```
1  renameElements(OldClassName,[],[],NewClassName).
2  renameElements(OldClassName,[PreElement|PreRest],[PostElement|PostRest],
       NewClassName) :-
3      '=..'(PreElement,[OldClassName|Arguments]),
4  renameElements(OldClassName,PreRest,PostRest,NewClassName),
5      '=..'(PostElement,[NewClassName|Arguments]).
6
7  renameRelations(OldAssocName,[],[],NewAssocName).
8  renameRelations(OldAssocName,[PreRelation|PreRest],[PostRelation|PostRest],
       NewAssocName) :-
9      '=..'(PreRelation,[OldCAssocName|Arguments]),
10 renameElements(OldClassName,PreRest,PostRest,NewAssocName),
11     '=..'(PostRelation,[NewAssocName|Arguments]).
```

B.2 (De-)Constructing The UML Models

```
1  migrate(UML1,UML2) :-
2      minuml1(UML1Pack,[],UML1Class,UML1Assoc), union([UML1Pack],[],UML1),
3      'Pseudostate'(UML1Pseudostate,UML1Pseudostates), 'FinalState'(
           UML1FinalState,UML1FinalStates),
4      'ActionState'(UML1ActionState,UML1ActionStates), 'Transition'(
           UML1Transition,UML1Transitions),
5      'Guard'(UML1Guard,UML1Guards), 'BooleanExpression'(UML1Expression,
           UML1Expressions),
6      'ObjectFlowState'(UML1ObjectFlowState,UML1ObjectFlowStates), 'Partition'
           (UML1Paritition,UML1Partitions),
7      'ActivityGraph'(UML1ActivityGraph,UML1ActivityGraphs),
8      union([UML1Pseudostate,UML1FinalState,UML1ActionState,UML1Transition,
           UML1Guard,UML1Expression,UML1ObjectFlowState,UML1Paritition,
           UML1ActivityGraph],RestUML1Class,UML1Class),
9      incoming(UML1incoming,UML1incomings), outgoing(UML1outgoing,
           UML1outgoings),
```

10 *expression*(*UML1expression*,*UML1expressions*), *subvertex*(*UML1subvertex*, *UML1subvertices*),
11 *contents*(*UML1content*,*UML1contents*), *transitions*(*UML1transition*, *UML1transitions*),
12 *guard*(*UML1guard*,*UML1guards*), *type*(*UML1type*,*UML1types*),
13 *partition*(*UML1partition*,*UML1partitions*),
14 union([*UML1incoming*,*UML1outgoing*,*UML1expression*,*UML1subvertex*, *UML1content*,*UML1transition*,*UML1guard*,*UML1type*,*UML1partition*], *RestUML1Assoc*,*UML1Assoc*),
15
16 migrateActivityGraph(*UML1ActivityGraphs*,*Activity*,*UML2Activities*),
17 migratePseudostate(*UML1Pseudostates*,*UML2InitialNodes*,*UML2ForkNodes*, *UML2JoinNodes*,*UML2DecisionNodes*),
18 renameElements('`FinalState`',*UML1FinalStates*,*UML2ActivityFinalNodes*,' `ActivityFinalNode`'),
19 renameElements('`ActionState`',*UML1ActionStates*,*UML2OpaqueActions*,' `OpaqueAction`'),
20 renameElements('`ObjectFlowState`',*UML1ObjectFlowStates*,*UML2ObjectNodes*,' `ObjectNode`'),
21 migrateTransition(*UML1Transitions*,*UML1ObjectFlowStates*,*UML1incomings*, *UML1outgoings*,*UML2ObjectFlows*,*UML2ControlFlows*),
22 migrateGuard(*UML1Guards*,*UML1Expressions*,*UML1expressions*, *UML2OpaqueExpressions*),
23 renameElements('`Partition`',*UML1Partitions*,*UML2ActivityPartitions*,' `ActivityPartition`'),
24 migratesubvertex(*UML1subvertices*,*UML1contents*,*Activity*,*UML2nodes*, *UML2Partnodes*),
25 renameRelations(*transitions*,*UML1transitions*,*UML2edges*,*edge*),
26 renameRelations(*partition*,*UML1partitions*,*UML2groups*,*group*),
27
28 '`Activity`'(*UML2Activity*,*UML2Activities*), '`ObjectFlow`'(*UML2ObjectFlow*, *UML2ObjectFlows*), '`ControlFlow`'(*UML2ControlFlow*,*UML2ControlFlows*), '`InitialNode`'(*UML2InitialNode*,*UML2InitialNodes*),
29 '`ForkNode`'(*UML2ForkNode*,*UML2ForkNodes*), '`JoinNode`'(*UML2JoinNode*, *UML2JoinNodes*), '`DecisionNode`'(*UML2DecisionNode*,*UML2DecisionNodes*),
30 '`ActivityFinalNode`'(*UML2FinalNode*,*UML2ActivityFinalNodes*), '`ActivityPartition`'(*UML2Partition*,*UML2ActivityPartitions*),
31 '`OpaqueExpression`'(*UML2OpaqueExpression*,*UML2OpaqueExpressions*), '`OpaqueAction`'(*UML2OpaqueAction*,*UML2OpaqueActions*), '`ObjectNode`'(*UML2ObjectNode*,*UML2ObjectNodes*),
32 union([*UML2Activity*,*UML2ObjectFlow*,*UML2ControlFlow*,*UML2InitialNode*, *UML2ForkNode*,*UML2JoinNode*,*UML2DecisionNode*,*UML2FinalNode*, *UML2Partition*,*UML2OpaqueExpression*,*UML2OpaqueAction*, *UML2ObjectNode*],[],*UML2Class*),
33 *node*(*UML2node*,*UML2nodes*), *edge*(*UML2edge*,*UML2edges*), *group*(*UML2group*, *UML2groups*), *nodes*(*UML2Partnode*,*UML2Partnodes*),
34 *edges*(*UML2Partedge*,[]), *outgoing*(*UML2outgoing*,*UML1outgoings*), *incoming*(*UML2incoming*,*UML1incomings*),
35 *guard*(*UML2guard*,*UML1guards*), *type*(*UML2type*,*UML1types*),

```
36    union([UML2node,UML2edge,UML2group,UML2Partnode,UML2Partedge,
          UML2outgoing,UML2incoming,UML2guard,UML2type],[],UML2Assoc),
37    minuml2(UML2Pack,[],UML2Class,UML2Assoc), union([UML2Pack],[],UML2).
```

B.3 Migrating Classes - Using Rename

```
1  migrateActivityGraph(UML1ActivityGraphs,Elem,UML2Activities) :-
2     'ActivityGraph'(ActivityGraph,Elem,Name), union([ActivityGraph],[],
          UML1ActivityGraphs),
3     'Activity'(Activity,Elem,Name), union([Activity],[],UML2Activities).
4
5  migratePseudostate(UML1Pseudostates,UML2InitialNodes,UML2ForkNodes,
       UML2JoinNodes,UML2DecisionNodes) :-
6     'Pseudostate'(Pseudostate,Elem,Name,initial), union([Pseudostate],RestStates,
          UML1Pseudostates),
7     migratePseudostate(RestStates,RestNodes,UML2ForkNodes,UML2JoinNodes,
          UML2DecisionNodes),
8     'InitialNode'(InitialNode,Elem,Name), union([InitialNode],RestNodes,
          UML2InitialNodes).
9  migratePseudostate(UML1Pseudostates,UML2InitialNodes,UML2ForkNodes,
       UML2JoinNodes,UML2DecisionNodes) :-
10    'Pseudostate'(Pseudostate,Elem,Name,fork), union([Pseudostate],RestStates,
          UML1Pseudostates),
11    migratePseudostate(RestStates,UML2InitialNodes,RestNodes,UML2JoinNodes,
          UML2DecisionNodes),
12    'ForkNode'(ForkNode,Elem,Name), union([ForkNode],RestNodes,UML2ForkNodes
          ).
13 migratePseudostate(UML1Pseudostates,UML2InitialNodes,UML2ForkNodes,
       UML2JoinNodes,UML2DecisionNodes) :-
14    'Pseudostate'(Pseudostate,Elem,Name,join), union([Pseudostate],RestStates,
          UML1Pseudostates),
15    migratePseudostate(RestStates,UML2InitialNodes,UML2ForkNodes,RestNodes,
          UML2DecisionNodes),
16    'JoinNode'(JoinNode,Elem,Name), union([JoinNode],RestNodes,UML2JoinNodes)
          .
17 migratePseudostate(UML1Pseudostates,UML2InitialNodes,UML2ForkNodes,
       UML2JoinNodes,UML2DecisionNodes) :-
18    'Pseudostate'(Pseudostate,Elem,Name,junction), union([Pseudostate],RestStates,
          UML1Pseudostates),
19    migratePseudostate(RestStates,UML2InitialNodes,UML2ForkNodes,
          UML2JoinNodes,RestNodes),
20    'DecisionNode'(DecisionNode,Elem,Name), union([DecisionNode],RestNodes,
          UML2DecisionNodes).
21 migratePseudostate([],[],[],[],[]).
22
23 migrateGuard(UML1Guards,UML1Expressions,UML1expressions,
       UML2OpaqueExpressions) :-
24    'Guard'(Guard,GuardElem,Name), union([Guard],RestGuards,UML1Guards),
25    'BooleanExpression'(BooleanExpression,ExpressionElem,Language,Body),
          union([BooleanExpression],RestBooleanExpressions,UML1Expressions),
```

```
26      expression(ExpressionRel,GuardElem,ExpressionElem), union([ExpressionRel],
            RestExpressions,UML1expressions),
27      migrateGuard(RestGuards,RestBooleanExpressions,RestExpressions,
            RestOpaqueExpressions),
28      'OpaqueExpression'(OpaqueExpression,GuardElem,Language,Body), union([
            OpaqueExpression],RestOpaqueExpressions,UML2OpaqueExpressions).
29  migrateGuard([],[],[],[]).
30
31  migrateTransition(UML1Transitions,UML1ObjectFlowStates,UML1incoming,
        UML1outgoing,UML2ObjectFlows,UML2ControlFlows) :-
32      'Transition'(Transition,Elem,Name), union([Transition],RestTransitions,
            UML1Transitions),
33      incoming(TargetElem,Target,Elem), union([TargetElem],Restincoming,
            UML1incoming),
34      'ObjectFlowState'(ObjectFlowState,Target,_), union([ObjectFlowState],
            RestObjectFlowStates,UML1ObjectFlowStates),
35      migrateTransition(RestTransitions,UML1ObjectFlowStates,Restincoming,
            UML1outgoing,UML2RestObjectFlows,UML2ControlFlows),
36      'ObjectFlow'(ObjectFlow,Elem,Name), union([ObjectFlow],UML2RestObjectFlows
            ,UML2ObjectFlows).
37  migrateTransition(UML1Transitions,UML1ObjectFlowStates,UML1incoming,
        UML1outgoing,UML2ObjectFlows,UML2ControlFlows) :-
38      'Transition'(Transition,Elem,Name), union([Transition],RestTransitions,
            UML1Transitions),
39      outgoing(SourceElem,Source,Elem), union([SourceElem],Restoutgoing,
            UML1outgoing),
40      'ObjectFlowState'(ObjectFlowState,Source,_), union([ObjectFlowState],
            RestObjectFlowStates,UML1ObjectFlowStates),
41      migrateTransition(RestTransitions,UML1ObjectFlowStates,UML1incoming,
            Restoutgoing,UML2RestObjectFlows,UML2ControlFlows),
42      'ObjectFlow'(ObjectFlow,Elem,Name), union([ObjectFlow],UML2RestObjectFlows
            ,UML2ObjectFlows).
43  migrateTransition(UML1Transitions,UML1ObjectFlowStates,UML1incoming,
        UML1outgoing,UML2ObjectFlows,UML2ControlFlows) :-
44      'Transition'(Transition,Elem,Name), union([Transition],RestTransitions,
            UML1Transitions),
45      migrateTransition(RestTransitions,UML1ObjectFlowStates,UML1incoming,
            UML1outgoing,UML2ObjectFlows,RestControlFlows),
46      'ControlFlow'(ControlFlow,Elem,Name), union([ControlFlow],RestControlFlows,
            UML2ControlFlows).
47  migrateTransition([],UML1ObjectFlowStates,Incoming,Outgoing,[],[]).&#10;
```

B.4 Migrating Associations - Using Rename

```
1   migratesubvertex(UML1subvertex,UML1contents,Activity,UML2node,UML2nodes) :-
2       subvertex(Subvertex,State,Vertex), union([Subvertex],RestSubvertex,UML1subvertex
            ),
3       contents(Content,Partition,Vertex), union([Content],RestContents,UML1contents),
4       migratesubvertex(RestSubvertex,RestContents,Activity,UML2node,RestNodes),
5       nodes(Nodes,Partition,Vertex), union([Nodes],RestNodes,UML2nodes).
```

```
6  migratesubvertex(UML1subvertex,[],Activity,UML2node,UML2nodes) :-
7      subvertex(Subvertex,State,Vertex), union([Subvertex],RestSubvertex,UML1subvertex
           ),
8      migratesubvertex(RestSubvertex,[],Activity,RestNode,UML2nodes),
9      node(Node,Activity,Vertex), union([Node],RestNode,UML2node).
10 migratesubvertex([],[],Activity,[],[]).
```

Abstract and Concrete Syntax Migration of Instance Models

Antonio Cicchetti[1], Bart Meyers[2], and Manuel Wimmer[3]

[1] Mälardalen University, MRTC, Västerås, Sweden
antonio.cicchetti@mdh.se
[2] University of Antwerp, Belgium
bart.meyers@ua.ac.be
[3] Vienna University of Technology, Austria
wimmer@big.tuwien.ac.at

Abstract. In this paper, we present a solution for the TTC 2010 model migration case study. Firstly, we present a modular approach to migrate the instance models' abstract syntax. Secondly, the problem of co-evolution of diagrammatical information such as icon positions and bend points of edges is identified and a solution specific to this case study is presented. Our solution implemented using ATL and Java.

Key words: Metamodel evolution, model co-evolution, inplace model transformations, model merging

1 Goal and approach

This paper presents a solution to the TTC 2010[4] model migration case study. This case study explores the consequences an evolution of a modelling language can have with respect to its instance models. In particular, these models must be migrated so that (i) they conform to the new language, and (ii) their semantics are preserved. The presented language evolution is UML 1.4 Activity Graphs to UML Activity Diagrams 2.2. The main goal of the case study is shown in the diagram of Figure 1. In our approach, a structured migration process is pursued by first modelling the evolution Δ itself into manageable parts, followed by a migration of each part resulting in a migration transformation M. The explicit breakdown of Δ helps us in understanding the evolution and finding a correct, semantically preserving migration M.

After migrating a model m using M, the diagrammatical information of m is lost. Specific to this case study, this results in a model m' for which the original positions of the icons are lost. This makes the migrated models less readable, as people tend to arrange the model icons in a way that is natural to them. Therefore, we decided to migrate the concrete syntax information or diagram model as well. Figure 2 shows the new diagram that includes this second goal. Apart from migrating models m as in Figure 1, the diagram model m_{CS} must

[4] http://planet-research20.org/ttc2010/

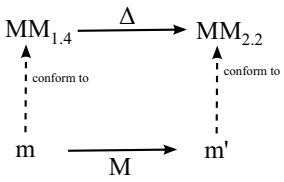


Fig. 1: The model migration case study with the evolution Δ and the migration M.

also be migrated by a migration transformation M_{CSinst}. The concrete syntax can be obtained by so-called rendering, which can be considered a transformation in which the user adds diagrammatical information, such as icons (at the level of the language concepts) and positions of these icons (at the level of the language concept instances). As suggested in Figure 2, we can make some assumptions for this case study: (i) the metamodels of the diagram model m_{CS} and its migrated counterpart m'_{CS} are the same: $MM_{GMF-notation}$ – GMF notation 1.0.2, and (ii) the rendering transformation itself must not be migrated, as in this case study, it is part of the GMF-based graphical editor which has evolved itself too (captured as the evolution Δ_{CS}, from $render_{1.4}$ to $render_{2.2}$). So in this case study, the migration of the concrete syntax MM_{CSinst} can be derived from the evolution Δ and Δ_{CS} (which in practice evolved together), and does not have to take a change of the metamodel $MM_{GMF-notations}$ into account. In conclusion, in this case study the concrete syntax co-evolution is simplified. However, this solution can start a discussion about the general topic of concrete syntax co-evolution.

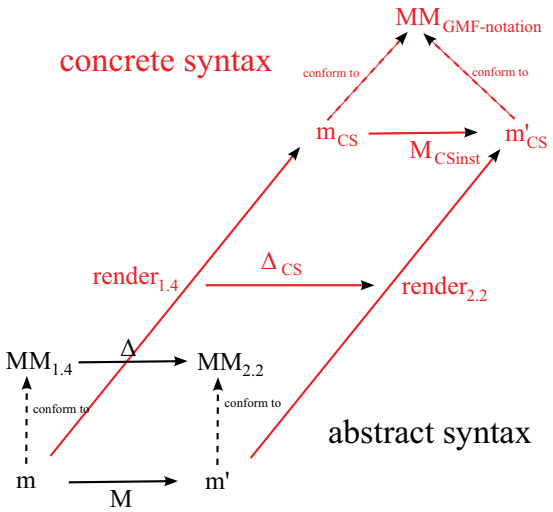


Fig. 2: MM_{CSinst}, The co-evolution of the concrete syntax instances.

2 Solution

In this section, we present an overview of our solution for the case study, consisting of the two parts: migration of instance models and migration of diagram models.

2.1 Instance Model Migration

As stated before, we structurally tackle the problem of instance model migration by starting off from the metamodel evolution and model co-evolution, as in the following:

metamodel evolution detection that is the old and new versions of the metamodel are compared in order to synthesise the evolution it has been subject to. This will result in a breakdown into manageable evolutionary steps;

modular migration creation for every evolutionary step, a migration activity is created (typically in the form of a transformation rule).

Our solution is based on the manual declaration of evolutionary steps; such choice guarantees that the intentions of the metamodel developer are fully captured. It is worth noting that the same result could be obtained by a tool recording the changes made by the user. In any case, in general the metamodels are remarkably smaller and more manageable than model instances, which makes even the manual specification of the evolution an acceptable effort if compared to verifying the migration correctness of existing instances.

We broke down the evolution into eleven evolutionary steps. For each step, the ATL migration rule is given. In the first two steps, a short explanation is given for the migration action; for the other rules, the migration is just a straightforward replacement of corresponding instances:

1. ActivityGraph, StateMachine, and CompositeState are merged into Activity. Subvertex and partition containers are merged into groups container to the new class ActivityGroup;
   migrate_TopCompositeState, *migrate_SubCompositeStates*
2. Transition becomes ActivityEdge with two subclasses, ControlFlow and ObjectFlow. In case of a surrounding ObjectFlowState, the transition becomes ObjectFlow, otherwise ControlFlow;
   migrate_Transition_ControlFlow, *migrate_Transition_ObjectFlow*
3. StateVertex and State are merged into ActivityNode;
   migrate_State
4. Pseudostate(kind:initial) becomes InitialNode;
   migrate_PseudoState_Initial
5. Pseudostate(kind:join) becomes JoinNode;
   migrate_PseudoState_Join
6. Pseudostate(kind:fork) becomes ForkNode;
   migrate_PseudoState_Fork

7. Pseudostate(kind:junction) is split into DecisionNode and MergeNode;
   migrate_PseudoState_Junction
8. FinalState becomes ActivityFinalNode;
   migrate_FinalState
9. ActionState becomes OpaqueAction;
   migrate_ActionState
10. Partition becomes ActivityPartition as subclass of ActivityGroup;
    migrate_Partitions
11. Guard becomes ValueSpecification and the contained BooleanExpression becomes the subclass of ValueSpecification called OpaqueExpression.
    migrate_Guard

2.2 Concrete Syntax Migration

The concrete syntax of UML models is not standardized like the abstract syntax. In our solution we used the Eclipse UML 2 tool suite[5], which includes a graphical editor for UML 2 models. The diagrammatical information this editor can read and write will be the target platform for migrating the concrete syntax of a model. On the other hand however, the Activity Graph 1.4 source models cannot be read by the UML 2 tool. Because we only want to use Eclipse, we built a simple UML 1.4 editor using GMF, that allows creating a model and position icons. The Activity Graph 1.4 example visualized by our simple editor is shown in Figure 3. After migration, the diagram model should be visualized in Eclipse as in Figure 4.

In order to obtain the correct result, the icon positions and bend points of edges must be preserved after migrating the abstract syntax as explained in Section 2.2. This requires a simple copying of the coordinates and bend points. However, it must be made sure that the copying of this information is done to the right element, a not obvious task given the exploitation of UUIDs. In other words, the graphical arrangement is saved in a XMI-like document by referring to model elements through UUIDs, whereas the ATL transformation is agnostic of them as working at the metamodel level. Therefore, trace information has to be created in order to link elements of the old diagram and the corresponding migrated ones of the new diagram. In particular, when the co-adapting ATL transformation is executed, a simple trace model is filled in with a list of (source, target) pairs storing the links between elements of the old and new metamodel instances.

In our solution, this is done by simply adding the code that creates the trace model in the ATL transformation. In fact, the addition of such code can also be done automatically, by using a transformation that adapts the ATL transformation. Because the input and output models of this transformation are transformations themselves, such a transformation is called a higher order transformation. Due to time limitations, we did not implement this higher order

[5] `http://www.eclipse.org/projects/project_summary.php?projectid=modeling.emf`

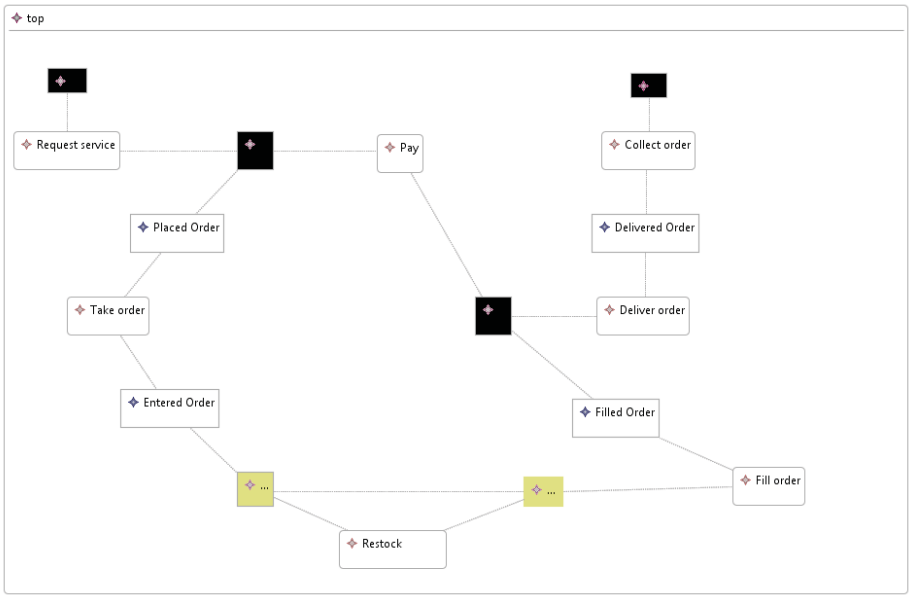


Fig. 3: The original model visualized by the simple GMF editor.

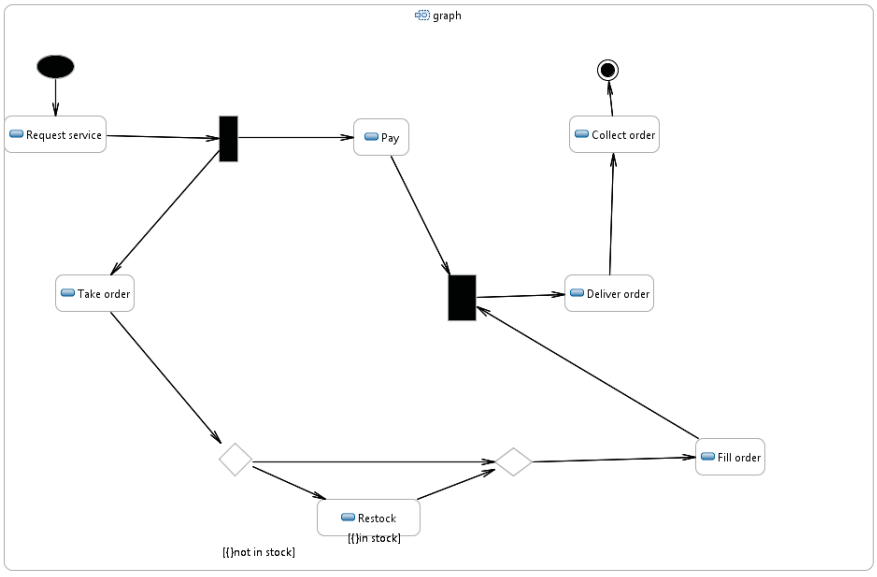


Fig. 4: The migrated model visualized by Eclipse UML 2.

transformation yet. It would improve automation, as well as clearly separating the code for creating the traceability model from the code for creating the migrated model. This lowers accidental complexity, increasing the overall quality (readability, maintainability, etc.) if the transformation models.

From the migrated instance model, the original instance model, the original diagram model and the trace model, the new diagram model can be automatically generated. For each element in the migrated instance model, its concrete syntax is generated by tracing back to the original element(s) using the trace model. The corresponding concrete syntax information which is obtained by using the `href` fields that point to the instance model elements are copied in the migrated diagram model. This approach is shown in Figure 5, where the links from the original diagram model to the migrated diagram model are shown from top to bottom.

3 Conclusion and Discussion

In our solution we presented a structured way of migrating the instance model by splitting up the evolution into manual parts. For every evolution step that emerges, (a) simple migration rule(s) can be created. The total migration transformation is implemented in ATL.

In order to keep the concrete syntax information of a model, the diagram model is also migrated. For this case study, this includes the preservation of icon positions and bend points. These can be simply preserved by copying them. However, in order to find the right elements for the copied information, a trace model is needed that links elements from the original instance model to the migrated instance model. This trace model is created by a part of the migration transformation.

The complexity of concrete syntax migration When looking at Figure 2, in this case study the concrete syntax migration was simplified in two ways: the concrete syntax metamodel remains the same (i.e., $MM_{GMF-notation}$) and the evolution of the editor does not have to be taken into account, as it is done manually. In the more general case of concrete syntax migration however, these assumptions cannot be made. In general, the migration of concrete syntax will entail two migration actions:

- the migration of the concrete instance models. We have done this for this case study. In the more general case however, the metamodels need not be the same. As a result, the diagram model will have to migrated across two dimensions: the conformance with the abstract syntax model, and the conformance with its metamodel;
- the migration of the rendering transformation. This can be considered the migration of the “editor”.

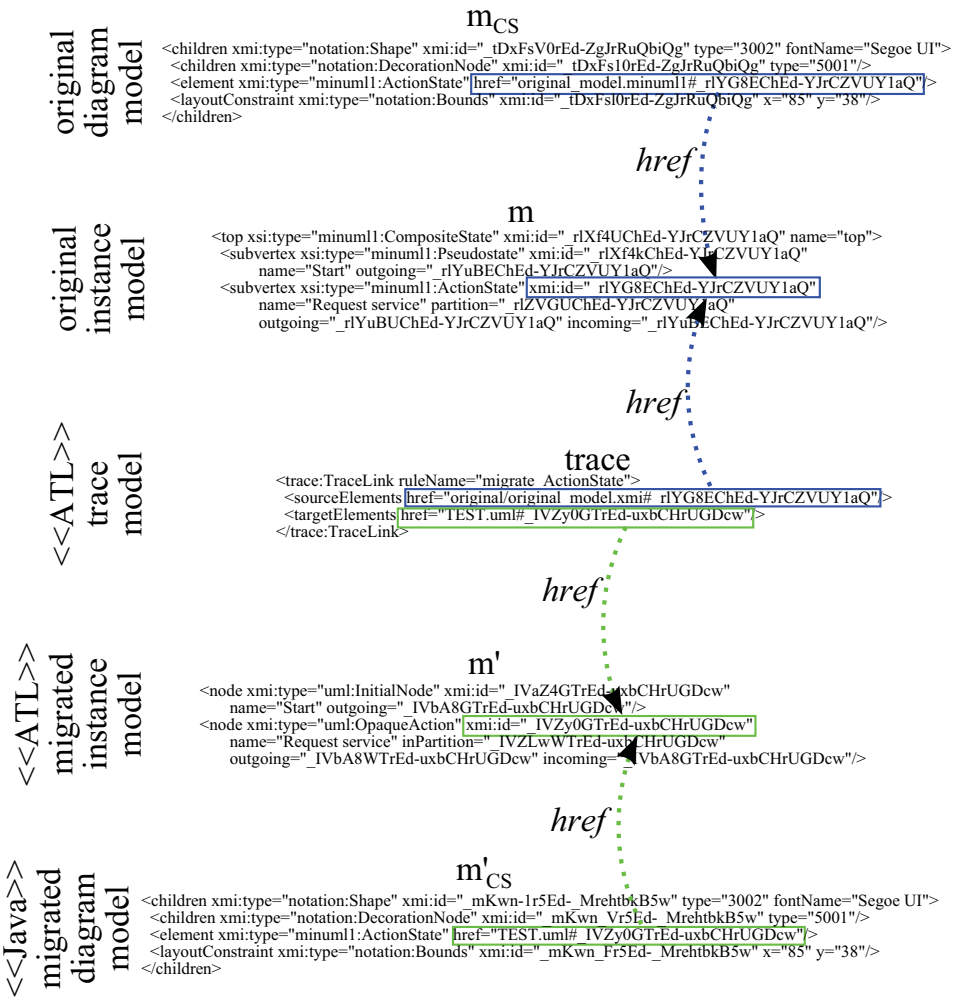


Fig. 5: The models are linked through trace links.

A Graph Transformation Case Study for the Topology Analysis of Dynamic Communication Systems*

Peter Backes[1] and Jan Reineke[2]

[1] Universität des Saarlandes, Saarbrücken, Germany
rtc@cs.uni-sb.de
[2] University of California, Berkeley
reineke@eecs.berkeley.edu

Abstract. We propose a case study for the Transformation Tool Contest 2010 that concerns dynamic communication systems (DCS). DCS are systems of autonomous processes that interact to achieve their goals. For this purpose, the processes exchange messages with each other. In contrast to distributed algorithms, the number of processes of the system is unbounded. The specific dynamic communication systems we want to investigate are so-called platoons. Platoons are groups of cars that drive on a highway with constant speed and constant distance to conserve energy. To form such platoons, each car follows the so-called merge protocol, which guides its local behaviour. We are interested in properties of the communication topologies that may emerge in this platoon scenario. Hence, we ask you to analyze a graph transformation system that generates the possible topologies of the merge protocol. The goal is to do this with as many processes as possible. The case study aims to improve understanding of how useful existing tools are for state space exploration and topology analysis.

1 Introduction

1.1 Context of the case

Dynamic communication systems are systems that have an unbounded and dynamically changing number of processes. Those processes communicate with each other in order to establish and transform communication topologies (see [2] for a more detailed description). In this case study, we want you to compute the topologies that may occur for the merge protocol, a communication protocol which is used in car platooning. Car platooning [4] concerns cars that drive on a highway with constant speed and constant distance from each other, to conserve energy.

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See http://www.avacs.org/ for more information.

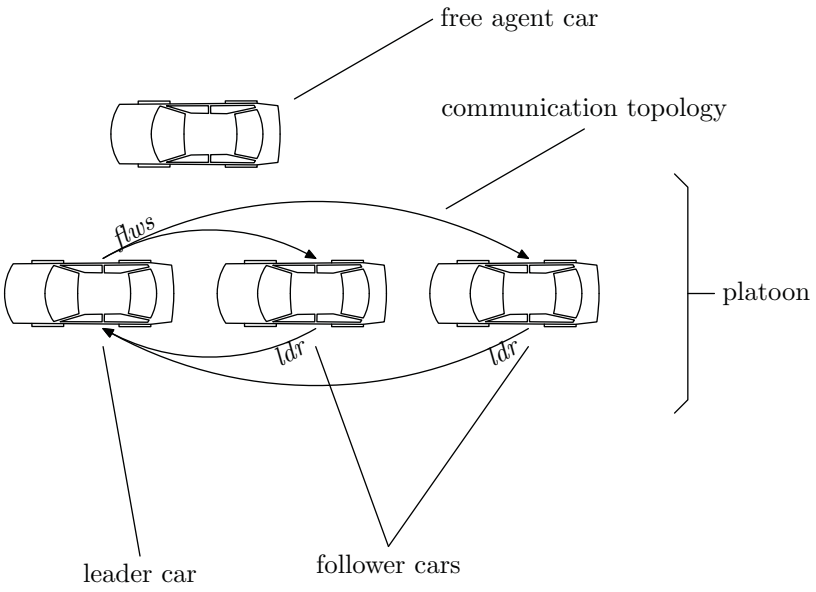


Fig. 1. Car platooning

The cars are equipped with wireless technology that allows them to communicate via messages. These messages are used to coordinate actions of the platoon, such as have new cars join the platoon or have the platoon change the lane. For this to work, one car per platoon acts as a leader of the platoon and the other cars—the followers—receive command messages from the leader so that the platoon as a whole acts in the desired manner. Accordingly, the platoon leader has to remember all its followers and each follower has to remember the leader. We call these relations among the cars the communication topology of the platoon.

The merge protocol describes how the cars use communication to join existing platoons and how two platoons manage to merge, so that only one platoon with one platoon leader is left once the merge has finished.

1.2 Purpose from a larger perspective

Our case study is supposed to shed light on how useful existing graph transformation tools are for network protocol and other concurrent systems analysis. We think that it has two major benefits:

1. It shows us how well existing graph transformation can do reachability analysis of network protocols, and to what number of processes they scale. Reachability analysis explores the state space of a network protocol and checks each state that is reachable for undesired properties; or, put differently, it

searches for bugs in the protocol. While more sophisticated tools are available for reachability analysis, they often require specialist knowledge about the inner workings of the respective algorithm. Graph transformation, on the other hand, is an intuitive and general approach that can be understood and used easily.
2. Reachability analysis of unbounded systems can only be partial, since it computes only a finite subset of a finite set of states. We still hope that results from such an analysis provide a good heuristic for the construction and evaluation of abstraction techniques for network protocol topology analysis like [3]. Such analyses serve two purposes: To directly verify safety properties—for example that two followers never assume each other to be their leader—and to provide invariants of the protocol that improve precision and efficiency of related analyses like [5].

1.3 Challenges that are involved

As the state space of the entire system grows rapidly with the number of processes, it is your goal to show that your analysis can scale well in that respect. This means that runtime and memory consumption should be kept as low as possible. How many processes can your tool handle? We suspect that it is possible only for a small single-digit number of processes.

The graph transformation system that implements the protocol to be analyzed uses rules with simple left hand sides (at most three nodes). During the state space exploration, many similar graphs arise that are matched by the same rules. Can your tool handle such cases efficiently?

2 The subject to be modelled

In this section, we describe the background of dynamic communication systems and the intuition behind the merge protocol. It is not necessary to understand all the details for the challenge—we will provide you with a set of graph transformation rules modelling the merge protocol.

2.1 Dynamic communication systems

Dynamic communication systems [1] consist of a finite but arbitrary number of processes that are in one of a finite set of states. Each process has a separate FIFO queue of unbounded length for messages from each of the other processes. Each message can optionally carry the identity of another process as a parameter, so processes can refer to other processes in their communication. Further, each process has a special queue for environment messages. Environment messages may be sent unconditionally by the environment, that is, they may be added to the queue at any time. They are necessary because they are the only way for disconnected parts of the system to get known to each other. In car platooning, for example, a sensor that is built in each of the cars might notice that another

car is in its communication range. As we abstract from the physical locations of cars, we model such sensors by the nondeterministic environment. Finally, each process locally maintains a finite set of channels. Each such channel holds a subset of the process identities of the entire system. Channels are a logical construct, not a physical one; they are rather like local address tables (if you assume that the cars use IPv6 to communicate and their identities are IP addresses), not like global wave frequencies shared by all processes; ie., Channels : $Id \to 2^{Id}$, not Channels $\subseteq 2^{Id}$. So two different processes may store different identities in the channels of the same name at the same time. A process communicates with other processes by sending a message to all processes in one of its channels. From a global view, the channels make up the communication topology.

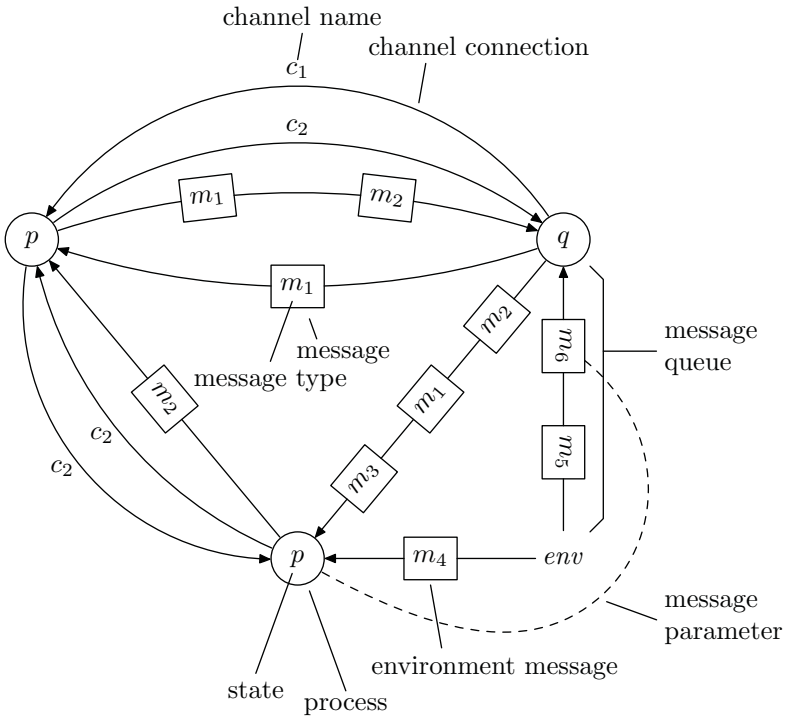


Fig. 2. Concepts of dynamic communication systems

A protocol specifies the behaviour of a process of a dynamic communication system. It consists of the states that the processes can be in and of the transitions among these states. Transitions are annotated with statements:

– $?(m, c, op)$ and $?(m)$ are guard statements and cause the respective transition to be executed only if a message of type m is at the front of one of the queues.

For the first version, the parameter of the message is to be combined with channel c by the set operation *op*. In both cases, the message that has been received is consumed from the respective queue. For example, $?(ca, ldr, =)$ consumes a *ca* ("car ahead") message from the queue, clears all process identities from its channel *ldr* ("leader") and stores the identity that was attached to the consumed *ca* message into that channel.

- $c = \emptyset$ is a guard, too, and allows a transition to be taken only if the channel c is empty.
- $!(m, c_1, c_2)$ sends a message of type m to all processes on channel c_1 and attaches the identity of one randomly chosen process on c_2 as a parameter. In case of c_2 being the special channel *id* ("identity"), the process attaches its own identity. For example, $!(req, ldr, id)$ sends a *req* message to the process(es) in channel *ldr* (the processes using the merge protocol happen to never store more than one identity in their *ldr* channel) and attaches the identity of the sending process itself attached as a parameter. $!(newf, ldr, flws)$ picks one of the identities from channel *flws* ("followers"), attaches it to a *newf* ("new follower") message and sends it to the process(es) from channel *ldr*.
- (c_1, op, c_2) combines c_1 and c_2 with the set operation *op* and saves the result in c_1 again. For example, $(bldr, \setminus, bldr)$ removes all identities from the channel *bldr* ("back leader").

A subset of the states, the initial states, specify in which states a new process may come into existence (initially with empty queues and channels).

2.2 The merge protocol

The merge protocol implements three tasks: Building a platoon out of two processes so that one of them becomes a leader and the other a follower; having a process join an existing platoon; and merging two platoons into one.

The merge protocol specifies that each process starts in the only initial state *fa* ("free agent") (denoted in 3 by the triangle). As can be seen in Figure 3, from this state, a process may essentially take two different routes: The upper one, to become a leader, or the lower one, to become a follower. The decision between these two options is made based on whether the process receives the environment message *ca* ("car ahead") with another process as a parameter, or whether it is the other way around and it is attached as a parameter to a car ahead message received by a different process. Let us assume the former happens. That means that it takes the lower route: It stores the attached process identity in its *ldr* ("leader") channel, sends back a *req* ("request") message to that process with its own identity as parameter and changes its state to *hon* ("hand over nothing"). The process that receives the request message will then take the upper route: It will receive the message in state *fa* and so add the identity of the process that sent the message, which it knows from the parameter of the *req* message, to its *flws* ("followers") channel and send back an *ack* ("acknowledgement") message with its own process identity. It then changes to state *ldb* ("leader"). As soon

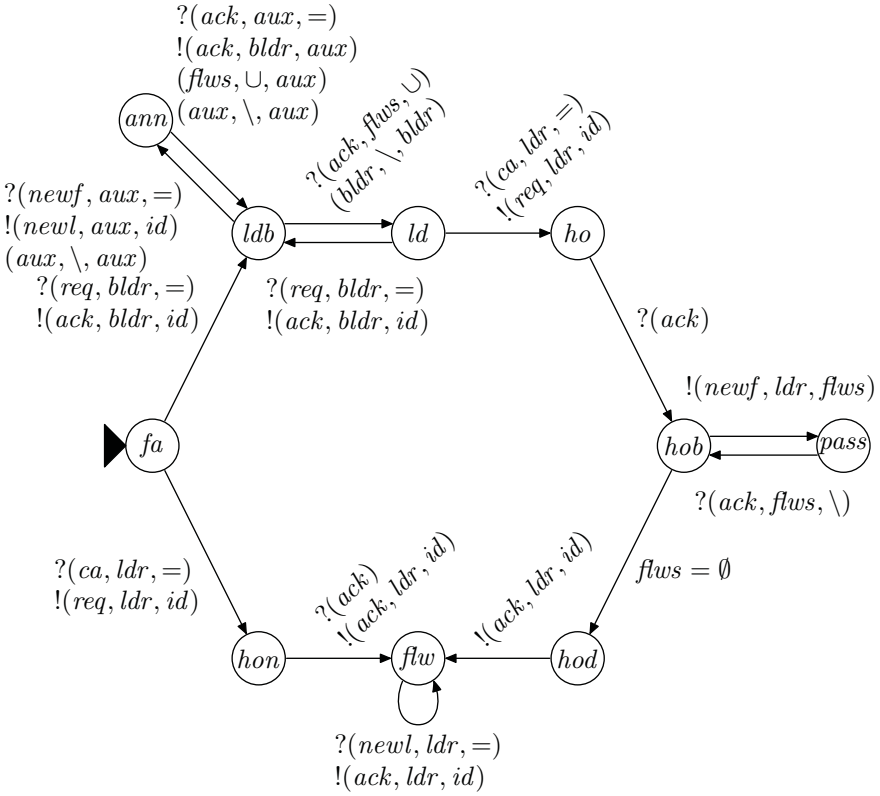
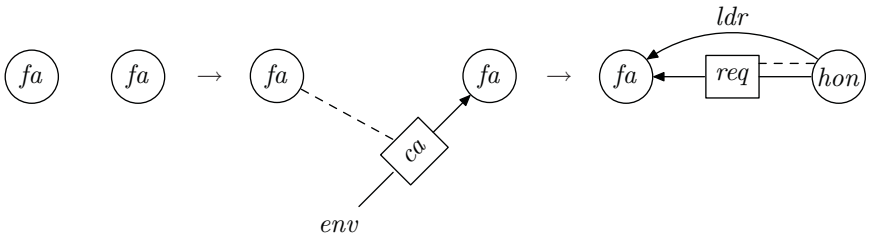
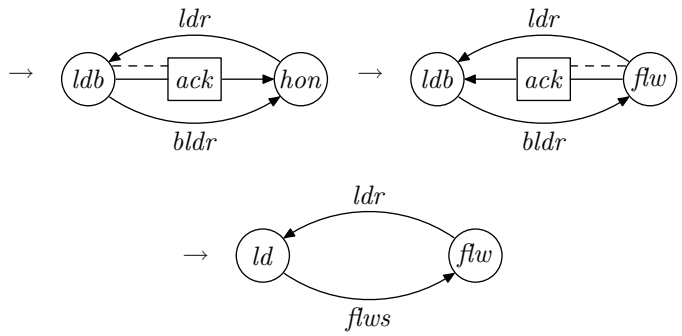


Fig. 3. Transition system of the merge protocol

as the other process, assuming it is still in state *hon*, receives the message, it changes to state *flw* ("follower") and returns another acknowledgement, which causes the other process to switch to state *ld* ("leader"). Once that has been done, the two cars have formed a platoon: The car that initially received the car ahead message has become a follower and the car that was attached as a parameter to that message has become the platoon leader. Here is a graphical representation of this evolution of the platoon:

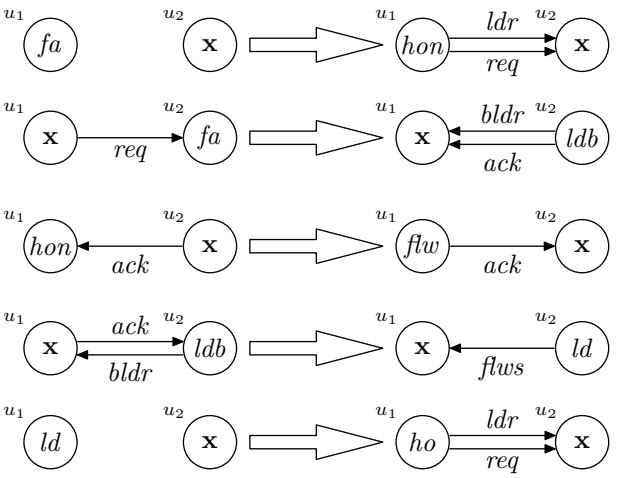


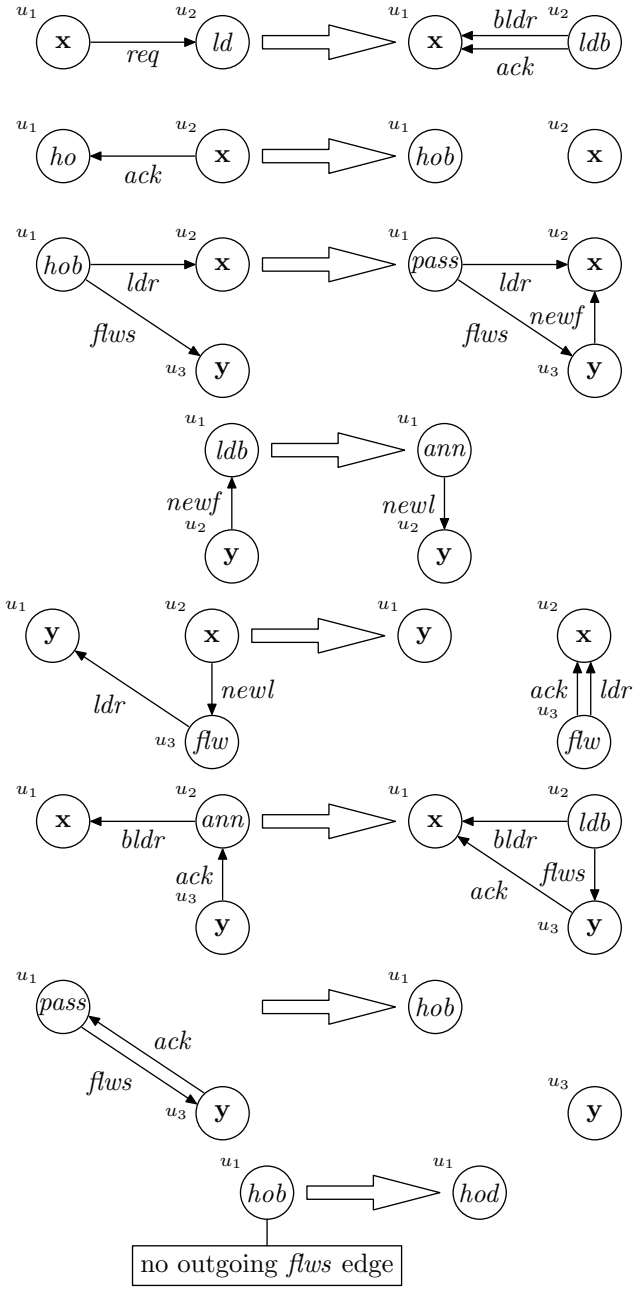


The second task, having a car join an already existing platoon, is mostly similar to the first one, except that the process attached to the car ahead message as a parameter is initially in state *ld* instead of *fa*. For merging (the third task), we have both the recipient of the car ahead message and the parameter process in state *ld*. Both enter a transmission loop to hand over the followers from the one leader to the other: The new platoon leader—the front leader—repeatedly switches between *ldb* (leader, expecting follower identities) and *ann* (waiting for acknowledgement after announcing itself as a new leader to a follower), the back leader repeatedly switches between *hob* (during handover, transmitting a follower identity) and *pass* (waiting for acknowledgement after passing a follower to the new leader), after having been temporarily in state in state *ho* (start of handover). Finally, the back leader goes to *hod* ("handover done") before becoming a follower itself. The auxiliary channel *aux* allows processes to temporarily store identites during transitions.

3 Implementation remarks

We provide a set of graph transformation rules (single pushout):




u1 x
u2 ld
req
u1 x
u2 ldb
bldr
ack
leader receiving merge or join request
u1 ho
u2 x
ack
u1 hob
u2 x
back leader entering transmission loop after acknowlegement of requested merge
u1 hob
u2 x
ldr
flws
u3 y
u1 pass
u2 x
ldr
newf
flws
u3 y
back leader notifying one of its followers about new leader
u1 ldb
u1 ann
newf
u2 y
newl
u2 y
front leader announcing itself to follower as the new leader
u1 y
u2 x
ldr
newl
u3 flw
u1 y
u2 x
ack
ldr
u3 flw
follower changing leader and acknowledging
u1 x
u2 ann
bldr
ack
u3 y
u1 x
u2 ldb
bldr
flws
ack
u3 y
front leader acknowledging handover of follower
u1 pass
ack
flws
u3 y
u1 hob
u3 y
back leader deleting link to follower just handed over
u1 hob
u1 hod
no outgoing flws edge
back leader finished transmission loop
u1 hod
u2 x
ldr
u1 flw
ldr
ack
u2 x
back leader acknowledging handover as completed and becoming a follower

These rules model the merge protocol in the following way:

- The configurations of the dynamic communication systems are modelled as graphs with node and edge labels
  - The node labels represent the state of the processes.
  - The edge labels represent the channels that make up the communication topology.
- The state transitions are modelled as graph transformation rules.
- Queues are represented as edges, and message types by edge labels (that are different from the labels used for the communication topology). Note that this is a simplification of the original system, as we disregard the order and exact number of messages in queues.
- **x** and **y** are variables. That mean that the corresponding nodes of the rule's left hand side should match nodes with arbitrary labels. All left hand side nodes with such variables in the rules above have corresponding right hand side nodes with exactly the same variable, which means that the rule application should not change the label of these nodes.
- The second to last rule, which implements the $\mathit{flws} = \emptyset$ transition, uses a negative application condition. The rule should only be applied to nodes with the label *hob* that do not have any outgoing edge with the label *flws*.

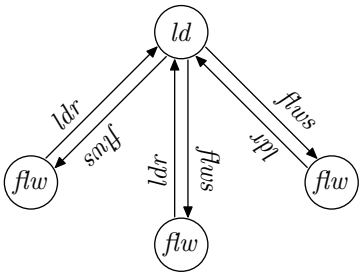
The rules use the following tricks:

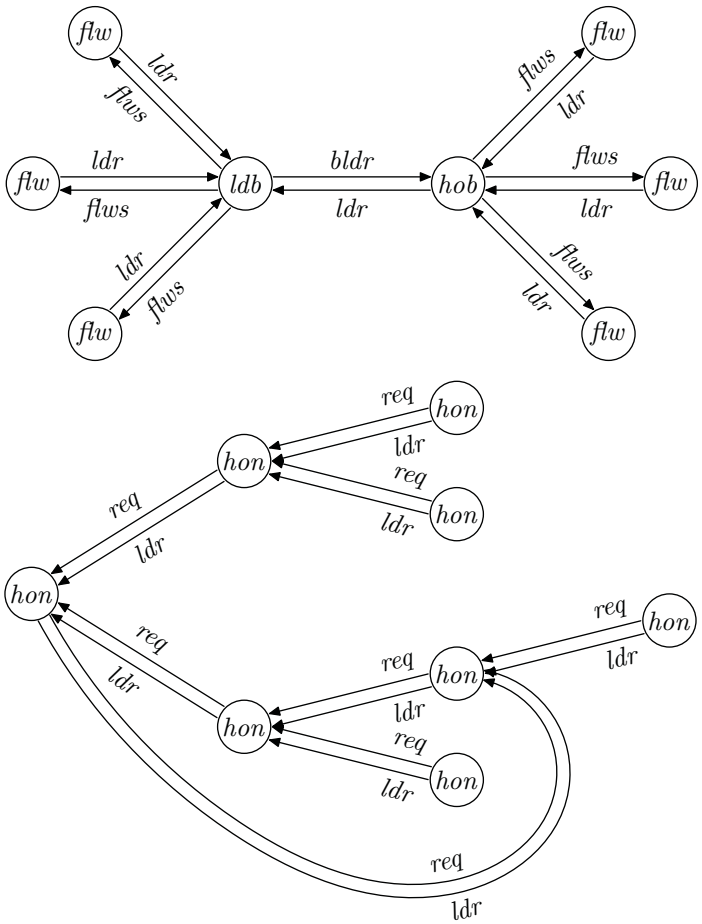
- We do not model environment messages. Because they can be sent at any time, they can also be received at any time.
- We do as many things as possible in one graph transformation rule.

In short, your tool should first read the transformation rules and the start graph. The start graph should consist of nodes that represent the processes, all in their initial state, i.e. *fa*, and with no edges. Then, the tool should compute by fixpoint iteration all reachable graphs of the graph transformation system. In each iteration, it has to find all matches of rules to any of the new graphs, apply the corresponding rule and add the result as a new graph (but only if no isomorphic graph has been computed before).

4 Example topologies

Here are some topologies that will emerge from the graph transformation system rules described above that model the merge protocol (assuming sufficiently many processes):





5 Goals

5.1 Core characteristics

- Output the topologies your tool computed. You may choose the output format freely.
- Feel free to cut down the problem to a reasonable size and ignore everything that you consider as an obstacle, even if you only analyze a small part of the protocol. The description of the DCS protocol in Section 2 should help you to adjust the graph transformation rules to your needs.

5.2 How the model should be used

- The graphs from the result are used for evaluating structural predicates on them, like "is there a node with label a and a node with label b such that an edge with label c points from the one to the other." Here is a list of properties that should be satisfied by the merge protocol:

  - No two nodes labelled *flw* are connected to each other with an edge labelled *ldr*.
  - A node labelled *pass* or *ld* always has at least one node labelled *flw* connected via some edge labelled *flws*.
  - If a node labelled *flw* has outgoing edges labelled *ldr* and *newf*, respectively, to two different nodes, then those two nodes are connected via an edge labelled *bldr*.

  How easy is it to evaluate such properties in your framework?
- The result should also be used for graphically displaying and exploring the topology structure of the platoons admitted by the protocol. Is it easy for the user to filter the displayed topology to eg. not include edges corresponding to messages?

5.3 Extensions

- We are also interested in a transition metagraph that models the evolution of the graph transformation system. It should have the graphs resulting from the analysis as nodes. Edges should be labelled with the respective rules that caused the transformation. This allows the inspection of traces.
- The graph transformation system we provide does not accurately reflect the DCS protocol with respect to message queues. Are you able to perform a queue analysis, either as part of the system analysis, or in a separate step, using graph transformation?
- Can you analyze the protocol in a general way using abstraction techniques such that the number of processes is not limited?

5.4 Evaluation criteria

We propose the following evaluation criteria:

- Completeness of the used transformation system: Less, same, more precise than reference transformation system? (more is better)
- Completeness of analysis: Systems with how many processes ($2 \ldots \infty$) were you able to analyze? (more is better)
- Performance: What is the memory consumption and runtime of the analysis for the largest analyzable system? (less is better)
- Flexibility of output: Do you merely allow output of topologies as they are, or do you allow filtering of edges/nodes according to labels, or even according to more complex filter specifications? (more is better)
- Flexibility of property evaluation: How powerful is your check for desired properties of topologies? Merely subgraph matching? More complex expressions over graphs with node and edge labels? (more is better)

References

1. Jörg Bauer, Ina Schaefer, Tobe Toben, and Bernd Westphal. Specification and verification of dynamic communication systems. In *Sixth International Conference on Application of Concurrency to System Design*, 2006.
2. Jörg Bauer and Reinhard Wilhelm. Static analysis of dynamic communication systems. In *14th International Static Analysis Symposium*, 2007.
3. Iovka Boneva, Arend Rensink, Marcos E. Kurban, and Jörg Bauer. Graph abstraction and abstract graph transformation. Technical Report TR-CTIT-07-50, Enschede, July 2007.
4. PATH Project. Vehicle platooning and automated highways. `http://www.path.berkeley.edu/PATH/Publications/Media/FactSheet/VPlatooning.pdf`, 1998.
5. Tobe Toben. Counterexample guided spotlight abstraction refinement. In K. Suzuki, T. Higashino, K. Yasumoto, and K. El-Fakih, editors, *Proceedings of the 28th IFIP WG6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2008)*, volume 5048 of *LNCS*, pages 21–36, Tokyo, Japan, June 2008. Springer-Verlag.

Solving the Topology Analysis Case Study with GROOVE

Amir Hossein Ghamarian Maarten de Mol Arend Rensink Eduardo Zambon

Department of Computer Science
University of Twente, The Netherlands
{ghamarian, molm, rensink, zambon}@cs.utwente.nl

1 Introduction

In this report we present our solution for the Topology Analysis case study of the Transformation Tool Contest (TTC) 2010 using the GROOVE tool set [4]. The case study is well within the scope of GROOVE, which allowed us to properly meet and sometimes surpass the goals set in the case description.

The case study addresses the problem of car platooning as an example of a dynamic communication system. The requested analysis concerns the so called *merge protocol* and the various communication topologies that may arise from it.

We begin by giving a brief explanation of the characteristics of the tool set in Section 2. In Section 3 we explain how the communication protocol was modelled in GROOVE and we present the experiments performed and the results obtained. An evaluation of the solution against the criteria given in the case description is presented in Section 4 and some concluding remarks are discussed in Section 5.

2 GROOVE

GROOVE is a general purpose graph transformation tool set that uses simple labelled graphs and single push-out (SPO) transformation rules. The core functionality of GROOVE is to recursively apply all rules from a predefined set (the graph production system – GPS) to a given start graph, and to all graphs generated by such applications. This results in a *state space* consisting of the generated graphs.

In this section we only present the features of GROOVE that are relevant for understanding the solution of the case study. For a more detailed description of GROOVE we refer the interested reader to [2] and the tool web site: `http://groove.cs.utwente.nl`. The entire tool set is written in Java and therefore it can be executed in any platform with a Java 6 virtual machine.

2.1 Tool Set

The GROOVE tool set has four components:

- **Editor.** An editor with a graphical interface for creating rules and host graphs.
- **Generator.** A command line tool that generates the state space of a GPS. The state space is stored as a Labelled Transition System (LTS), where each state is a graph and transitions are labelled by the rule applications. The strategy according to which the state space is explored (e.g., depth-first, breadth-first, etc) can be set as a parameter.

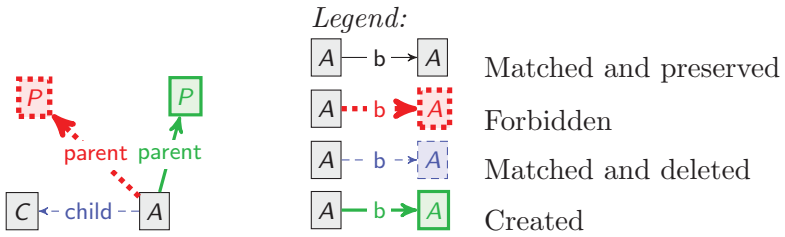


Figure 1: Example GROOVE rule and legend

- **Model Checker.** A command line tool that checks if properties expressed in temporal logic (CTL) hold in a LTS produced by the Generator. If a property does not hold, a counter-example is given.
- **Simulator.** A graphical interface tool that integrates the functionalities of the Editor, Generator, and Model Checker. In addition, the Simulator allows the user to interactive explore the LTS, by manually applying rules to a host graph.

2.2 Host Graphs

In GROOVE, the host graphs, i.e., the graphs to be transformed, are simple graphs with labelled nodes and edges. In simple graphs, edges do not have an identity, and therefore parallel edges (i.e., edges with same label, and source and target nodes) are not allowed.

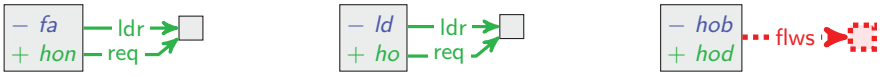
In the graphical representation, nodes are depicted as rectangles and edges as binary arrows between two nodes. Node labels can either be node types or flags; the former is not used in the solution and will not be discussed further. Flags are used to model a boolean condition, which is true for a node if the flag is there and false if it is absent. Flags are displayed in *italic* inside a node rectangle.

2.3 Rules

The transformation rules in GROOVE use the single push-out approach. The left and right hand side of a rule are combined in a single graph and colours and shapes are used to distinguish different elements. Fig. 1 shows a small example rule.

- **Readers.** The black (continuous thin) nodes and edges must be present in the host graph for the rule to be applicable and are preserved by the rule application;
- **Embargoes.** The red (dashed fat) nodes and edges must be absent in the host graph for the rule to be applicable;
- **Erasers.** The blue (dashed thin) nodes and edges must be present in the host graph for the rule to be applicable and are deleted by the rule application;
- **Creators.** The green (continuous fat) nodes and edges are created by the rule application.

Embargo elements are usually called Negative Application Conditions (NACs). When a flag is used in a non-reader element but the node itself is not modified, the flag is prefixed with a character to indicate its role. The characters used are $+$, $-$, and !, respectively for creator, eraser, and embargo elements.



(a) Rule fa-hon (b) Rule ld-ho (c) Rule hob-hod

Figure 2: Sample rules of the solution with injective matching

3 Solution

The reference transformation system given in the case description was simple to model in GROOVE. Each node of the host graph represents a car, and a flag indicates the car state in the protocol. Labelled edges represent the communication topology between cars. All the cars in a host graph start at the *fa* (free agent) state.

There is a trivial one-to-one mapping from the 14 rules given in the reference transformation system and the rules implemented in GROOVE. Since the reference rules are also in the SPO approach, the only translation necessary was to merge the left and right hand side of the reference rules into the single graph format of GROOVE. As expected, elements occurring only in the left [right] hand side of a reference rule became eraser [creator] elements in the corresponding GROOVE rule; and the intersection of left and right hand side of a reference rule defines the reader elements in the GROOVE rule.

Fig. 2 show the corresponding GROOVE rules of reference rules 1, 5 and 13, respectively. To give them a more meaningful identifier, GROOVE rules are named by the modification in the node state, e.g., fa-hon. The matching of nodes with arbitrary labels, described in the case description with **x** and **y** variables, is done in GROOVE with unlabelled nodes, e.g., Fig. 2(a).

An interesting point to note is that the default rule matching in GROOVE is non-injective whereas in the reference system the rules are clearly injective. This default behaviour in GROOVE can be changed by setting the option "Match Injective" to "true" in the grammar properties. Another possibility is to add explicit injectivity constrains (as NACs) to rules when necessary. We performed experiments with both variations.

3.1 Structural predicates on graphs

As explained in the case description, the point of computing the communication topologies of the protocol is to allow the evaluation of structural predicates on the topologies. These structural predicates express properties that should be satisfied by the merge protocol.

The case study describes three structural predicates. These can be modelled in GROOVE using *inspection rules*, i.e., rules with just reader and embargo elements. Fig. 3 shows the three inspection rules used in our solution. In the verification of the protocol we are interested in checking for the absence of an undesired property, therefore the inspection rules are the negation of the structural predicates described in the case study. Take, for example, the first predicate that states that "no two nodes labelled *flw* are connected to each other with an edge labelled *ldr*". Rule flw-ldr matches *any* two nodes labelled *flw* that are connected by an edge labelled *ldr*. Therefore, if we ever find a match for this rule then the original predicate was violated.

3.2 Experiments

We created host graphs from two to twelve cars. The exploration of the state space for graphs with at most six cars can be comfortably performed in the Simulator. For seven cars or more the state space becomes too large and the use of a GUI is no longer adequate.

In order to analyse the scalability of the solution we performed experiments that used only the command line tools, i.e., the Generator and the Model Checker. After producing the state

(a) Rule flw-ldr (b) Rule ld-flw (c) Rule no-bldr

Figure 3: Inspection rules for evaluating structural predicates

Table 1: Results for the first experiments performed. Breadth-first exploration with injective rule matching and isomorphism reduction on.

Cars	States	Transitions	Time (s)	Space (MB)
2	7	7	< 1	1
3	32	53	< 1	1
4	154	353	< 1	1
5	705	2,102	< 1	2
6	3,329	12,135	2	2
7	15,473	67,244	7	6
8	72,434	364,987	30	26
9	338,130	1,942,808	137	1,221
10	1,580,449	10,200,436	703	1,737

space with the Generator, we used the Model Checker to analyse the following CTL formula: AG(!flw-ldr & !ld-flw & !no-bldr). This formula expresses the property that on all generated states no match for any of the inspection rules is found. In all our experiments the Model Checker gave a *pass* verdict, meaning that the desired structural properties are satisfied.

The experiments were performed in a Dual Quad-Core Intel Xeon X5365 3.0 GHz machine with 32 GB of RAM, running Linux 2.6.31 and Java VM 1.6.13, both 64-bit versions. The initial results obtained are given in Table 1. The exploration strategy used was breadth-first, the rule matching was injective and the reduction of the LTS modulo isomorphism was on. The time given in Table 1 is the total running time of the Generator plus Model Checker. Less than 1% of this time is used for checking the formula; as expected the bulk of the running time is spent on generating the state space. The column **Space** shows the amount of memory used at the end of execution. Note that during execution the memory consumption may be higher.

There are several options and modelling choices that may affect the performance of GROOVE for a certain GPS. The first point that we investigated was the matching of rules. The initial experiments used injective matching and in order to revert back to the default GROOVE behaviour, i.e., non-injective matching, we had to modify two rules, namely fa-hon and ld-ho. The nodes in the left-hand side of these rules are not connected by any edge, so it was necessary to add a constraint indicating that the nodes must be distinct. This is done in GROOVE with a NAC edge labelled "=". Fig. 4 shows the modified rules. The memory consumption was unaltered by this change and the execution time decreased, but only slightly (less than 5%).

We also analysed how different exploration strategies may affect performance. We considered breadth-first (BFS) and depth-first (DFS) exploration. The final results are given in Table 2.

(a) Rule fa-hon (b) Rule ld-ho

Figure 4: Modified rules of the solution with non-injective matching

Table 2: Final results of the experiments performed with non-injective rule matching and isomorphism reduction on.

			BFS		DFS	
Cars	**States**	**Transitions**	**Time (s)**	**Space (MB)**	**Time (s)**	**Space (MB)**
2	7	7	< 1	1	< 1	1
3	32	53	< 1	1	< 1	1
4	154	353	< 1	1	< 1	1
5	705	2,102	< 1	2	< 1	2
6	3,329	12,135	2	2	2	2
7	15,473	67,244	6	6	6	6
8	72,434	364,987	30	26	26	25
9	338,130	1,942,808	132	1,221	166	125
10	1,580,449	10,200,436	692	1,748	7,629	623
11	7,383,773	52,931,771	29,270	4,359	79,835	4,269
12			out of memory		aborted	

Table 3: Comparison of state space sizes with and without isomorphism reduction.

		Cars										
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
States	**Iso On**	7	32	154	705	3,329	15,473	72,434	338,130	1,580,449	7,383,773	†
	Iso Off	12	174	3,104	68,900	1,838,052	‡					

† Out of memory after exploring 10,154,872 states and 95,745,200 transitions.
‡ Out of memory after exploring 6,278,576 states and 28,651,500 transitions.

It was possible to verify the protocol for graphs with up to eleven cars, which is more than the small single-digit number expected by the authors of the case study. The differences between the exploration strategies become more apparent for systems with nine or more cars. BFS is faster than DFS but consumes more memory. This shows the standard trade-off between memory consumption and execution time. In any case, it was not possible to fully explore the state space of the system with twelve cars. BFS exploration ran out of memory and DFS was aborted after running for several hours.

One interesting point to note is that the collapsing of states under isomorphism leads to a big reduction of the state space. We performed tests with and without isomorphism reduction to compare the size of the state spaces. The results are given in Table 3 and in the accompanying plot in Fig. 5. Despite being a computationally expensive operation, isomorphism checking pays-off when used in cases with a lot of symmetry, such as this one. In this case study, nearly 90% of the execution time is spent in isomorphism checking. Nevertheless, the reduction on the state space provided by this analysis still outweights the cost of performing isomorphism checks and allows us to increase the size of the systems that are verified. This is one advantage of GROOVE over non-graph based exploration tools that cannot benefit from this method of symmetry reduction [1].

4 Evaluation

In this section we go through the goals described in the case study and evaluate how GROOVE and our solution fare with respect to these points.

Core characteristics.
It was not necessary to simplify the proposed problem. GROOVE has all the features necessary to model the solution. All the states in the state space of the transformation system (i.e., the topologies) can be generated and stored using the option "Export Simulation" in the Simulator. Of course, saving the entire state space becomes unfeasible for larger inputs. The output state

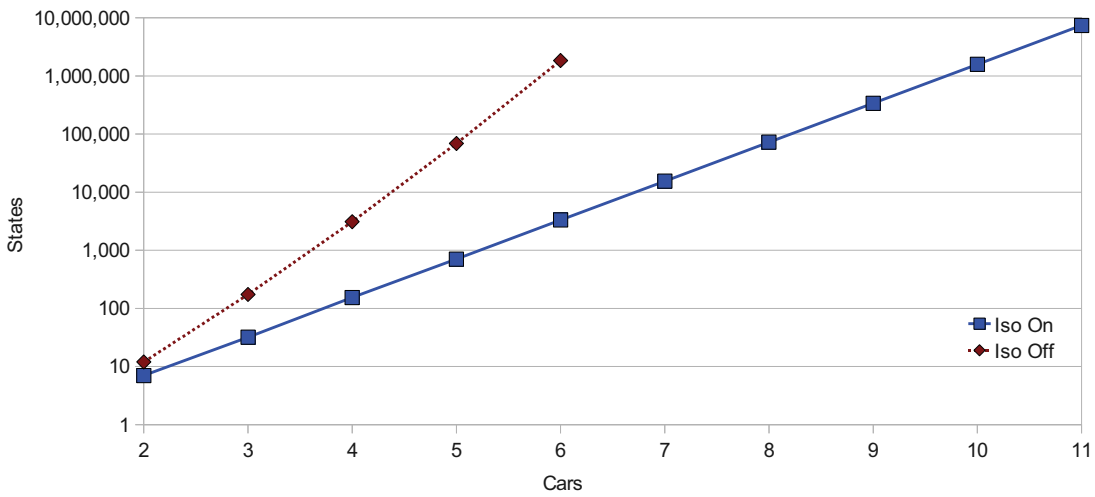


Figure 5: A log-plot of the state space size versus the numbers of cars

Figure 6: An example topology graph with some edges grayed out

graphs are saved in the GXL format.

How the model should be used.
The structural properties described in the case study were properly modelled by inspection rules. It is very simple to evaluate such properties in GROOVE. In Section 3 we described the non-interactive way of doing this evaluation using the Model Checker and CTL formulae. There is also an interactive way by means of the Simulator, which allows for a graphically visualisation of the topologies and an interactive application of rules. The Simulator has several visualisation capabilities, such as zooming and filtering of nodes and edges based on labels. Fig. 6 shows an example of a topology with six cars where message edges are grayed out. Alternatively, any element can be removed altogether from the view.

Extensions.
The so-called "transition meta-graph" described in the case study corresponds to the LTS in GROOVE terminology. Again, the Simulator has several interactive capabilities that allow the inspection of traces. Fig. 7 presents the LTS for a system with three cars. States marked in red correspond to dead-locked topologies.

The extension regarding the analysis of message queues was not considered in our solution.

The extension regarding the use of abstraction to allow the analysis of systems with an unlimited number of cars was not considered in our solution.

5 Conclusion

To conclude and to ease the comparison between solutions we sum up the evaluation criteria given in the case description.

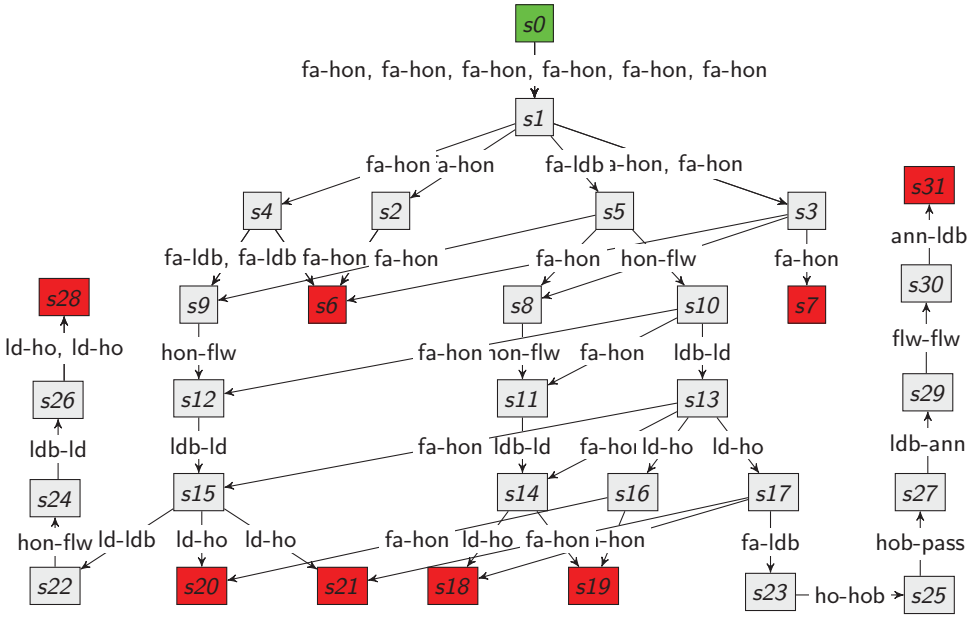


Figure 7: The LTS for a system with three cars

- **Completeness of the used transformation system.** We implemented the same reference transformation system.
- **Completeness of analysis.** We were able to analyse systems with up to 11 processes.
- **Performance.** For the largest analysable system the run-time was 8.1 hours and the memory consumption was 4.4 GB (using BFS exploration).
- **Flexibility of output.** Node and edges can be filtered according to labels.
- **Flexibility of property evaluation.** GROOVE allows nested quantification of rules, which has the same expressivity as First-Order Logic [3]. Furthermore, wildcards and regular expressions can be used to express more complex matching conditions of inspection rules.

The solution for the case study could be properly developed in GROOVE with very little effort and time. Since this case stresses performance aspects of the tool, it is interesting to highlight the points of improvements planned.

While profiling the Generator to identify performance bottle-necks in this case study, we established that roughly 90% of the execution time is spent in isomorphism checking. To address this we are currently investigating the use of canonical forms of graphs for isomorphism comparison. In large state spaces this is expected to yield better performance results.

Another important point for improvement is rule matching. We are working on the implementation of a RETE algorithm that allows for the use of incremental pattern matching of rules into the host graph. Since the rules are simple and small, this new algorithm is expected to give a performance boost.

The GPS of this case study has one interesting characteristic: it does not delete or create nodes. All algorithms in GROOVE are designed to handle addition and removal of nodes and thus cannot explore this particular characteristic of the case to improve the performance. It would be interesting to check how much can be gained with a dedicated solution to this case.

References

[1] P. Crouzen, J. C. van de Pol, and A. Rensink. Applying formal methods to gossiping networks with mCRL and GROOVE. *ACM SIGMETRICS performance evaluation review*, 36(3):7–16, December 2008.

[2] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. To appear, June 2010.

[3] A. Rensink. Representing first-order logic using graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335, Berlin, 2004. Springer Verlag.

[4] Arend Rensink. The GROOVE simulator: A tool for state space generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, (AGTIVE)*, volume 3062 of *LNCS*, pages 479–485. Springer, 2004. See `http://sourceforge.net/projects/groove`.

Abstract topology analysis of the join phase of the merge protocol*

Peter Backes[1] and Jan Reineke[2]

[1] Universität des Saarlandes, Saarbrücken, Germany
rtc@cs.uni-sb.de
[2] University of California, Berkeley
reineke@eecs.berkeley.edu

Abstract. We present a partial solution to the TTC2010 topology analysis case study. We pick a small part of the merge protocol, namely the part where cars join a leader to form a platoon. Using abstract interpretation, we compute an approximation of the arising topologies, without limiting the number of cars.

1 Introduction

In our case study, we ask "Can you analyze the protocol in a general way using abstraction techniques such that the number of processes is not limited?" In this solution to the case study, we achieve this goal for a part of the protocol. Section 2.2 of the case study mentions that the protocol implements three tasks. The part that we analyze consists of the first and the second task: Building a platoon out of two processes so that one of them becomes a leader and the other a follower, and having a process join an existing platoon.

If we apply the merge protocol graph transformation rules from the case study to a start graph with finitely many nodes, then they produce a finite set of graphs. This is so because none of the rules adds new nodes. If, on the other hand, we use an empty start graph, and add a new rule that merely creates free agent nodes, the result will be an infinite set of graphs. Accordingly, such a system cannot be analyzed using classic graph transformation tools.

Abstract interpretation allows us to compute approximations of such systems. The state of the art technique using this paradigm is partner abstraction [2,3], implemented in the tool `hiralysis`. However, partner abstraction was designed

* This work was supported by the DFG as part of the Transregional Collaborative Research Center SFB/TR 14 AVACS. In addition, it was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota.

for simple topologies only and requires a human expert to supply additional invariants—partner constraints—to cut off parts of the merge protocol that involve more complicated topologies. Such more complicated topologies do occur in the merge protocol [1]. Without cutting off these parts, partner abstraction will suffer from state space explosion, and `hiralysis` will not terminate.

The more complicated topologies cut off for partner abstraction analysis already arise for the two mentioned tasks. We aim at an abstraction that does not need manual intervention to cope with these topologies.

In section 2, we introduce our abstraction. Section 3 presents which parts of the protocol we analyze and the abstract result that we get after running `astra`, our analysis tool. We talk about the evaluation of properties on the abstract result in section 4. Section 5 sums up our work and presents future research.

2 Star abstraction

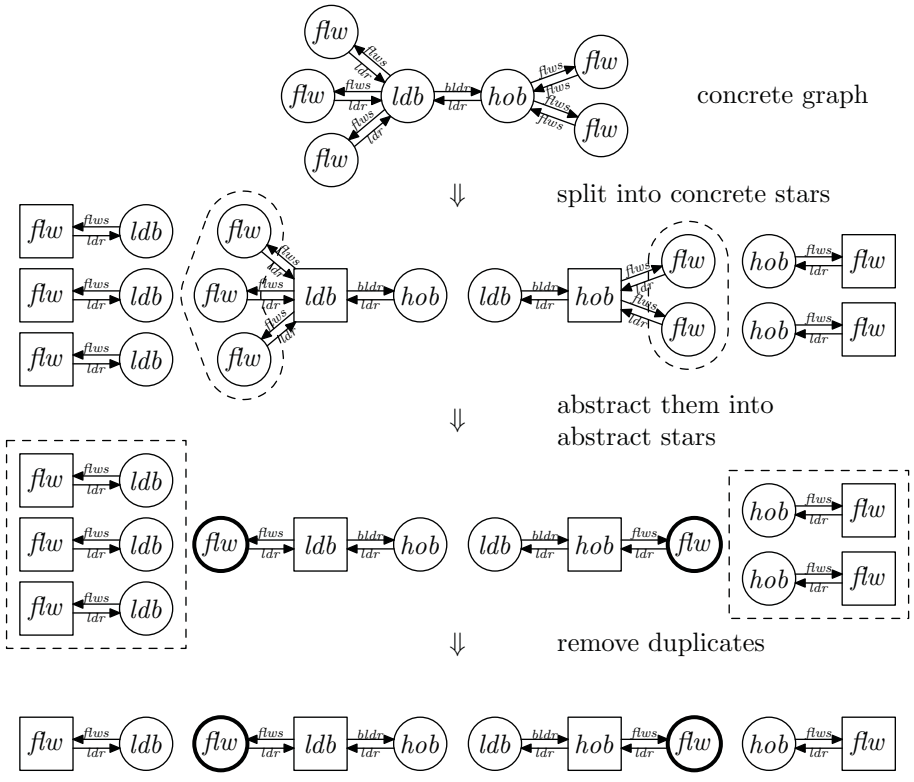


Fig. 1. Our abstraction applied step by step to a simple example

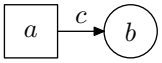
Our abstraction is sketched in Figure 1. We abstract graphs in three steps: First, we build the corresponding *star* for all nodes v of the graph. We obtain the star by removing all nodes from the graph except for v and its partner nodes, and by removing all the edges that are not incident to v. We call v the *core node* (displayed as a square in Figure 1) and the other nodes the *outer nodes*.

The next step is done for each star separately. We identify sets of outer nodes that cannot be distinguished from each other with respect to their label and the labels and directions of the edges incident to them. For each such set that contains two or more indistinguishable nodes, we merge them all into a summary node. We are then left with *abstract stars*. We represent the abstract stars as a tuple (l, E, A, S) with $l \in \mathcal{N}$ being the label of the core node, $E \subseteq \mathcal{E}$ being the self-loops of the core node, $A \subseteq \mathcal{N} \times 2^{\mathcal{E}} \times 2^{\mathcal{E}}$ being the *axes* and $S \subseteq A$ specifying which of those axes have summary nodes. Each axis (l, in, out) represents an outer node with label l and the edges incident to it. *in* contains those edges that point from the core node to the outer node and *out* the remaining ones. By construction, it follows that at least one connection must exist, that is, $in \cup out \neq \emptyset$.

In the final step, we ensure that each abstract star is unique. This is accomplished by keeping at most one copy from each class of isomorphic abstract stars.

There are $|\mathcal{N}| \cdot 2^{|\mathcal{E}|} \cdot 3^{|\mathcal{N}| \cdot (2^{2 \cdot |\mathcal{E}|} - 1)}$ different stars: Each star can have $|\mathcal{N}|$ different node labels for the core node and can have any subset of the $|\mathcal{E}|$ edge labels for self-loops. Between the core node and each outer node, there can be any edge with one of the $|\mathcal{E}|$ edge labels, either pointing from the core node to the outer node, or the other way around; with the exception that in total, at least one edge must be present. The outer node of an axis can have any of the $|\mathcal{N}|$ node labels, and each axis is either present, absent or present as a summary axis. We denote the set of all possible stars over a node label set $\mathcal{N}$ and an edge label set $\mathcal{E}$ as $\mathfrak{S}(\mathcal{N}, \mathcal{E})$.

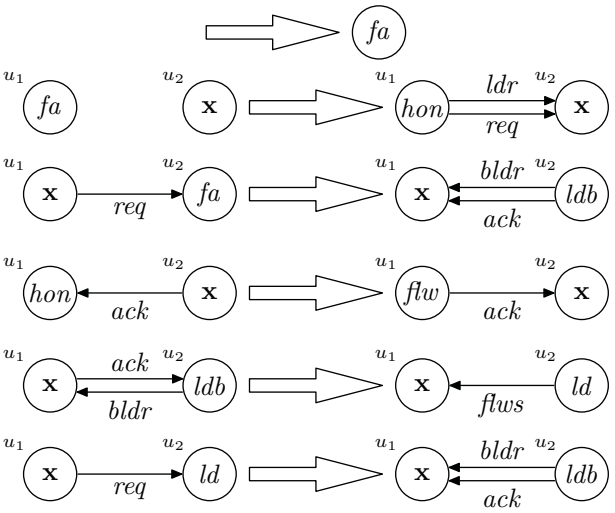
We do not yet have a result on the exact size of the star domain itself and merely know that it is bounded by $2^{|\mathfrak{S}(\mathcal{N},\mathcal{E})|}$. It is non-trivial, because not every subset of the stars is a valid abstraction of an actually existing concrete graph. For example, a set containing only the star



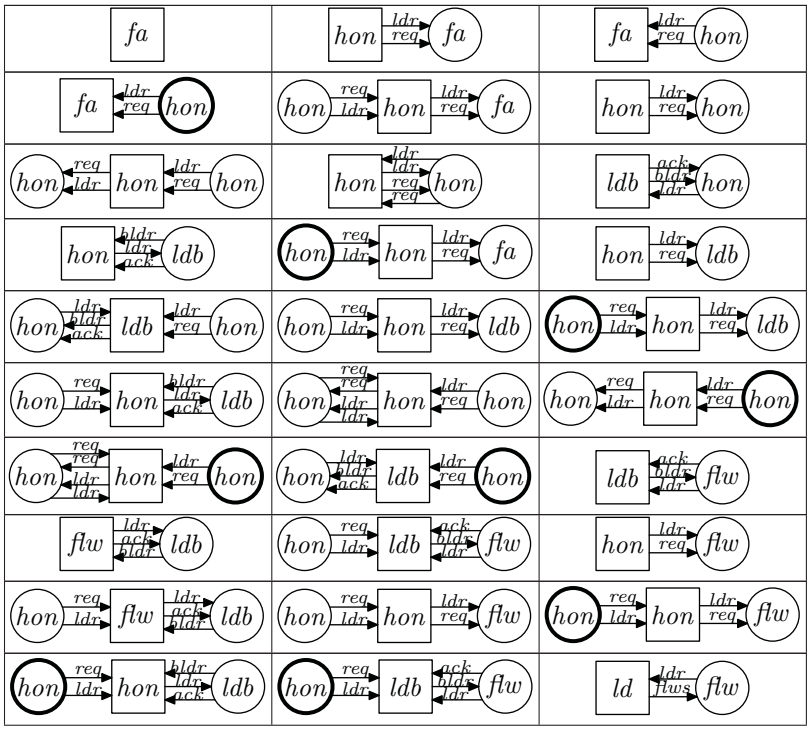
is not a abstraction of any graph, since there is no corresponding star with a core node that has label b.

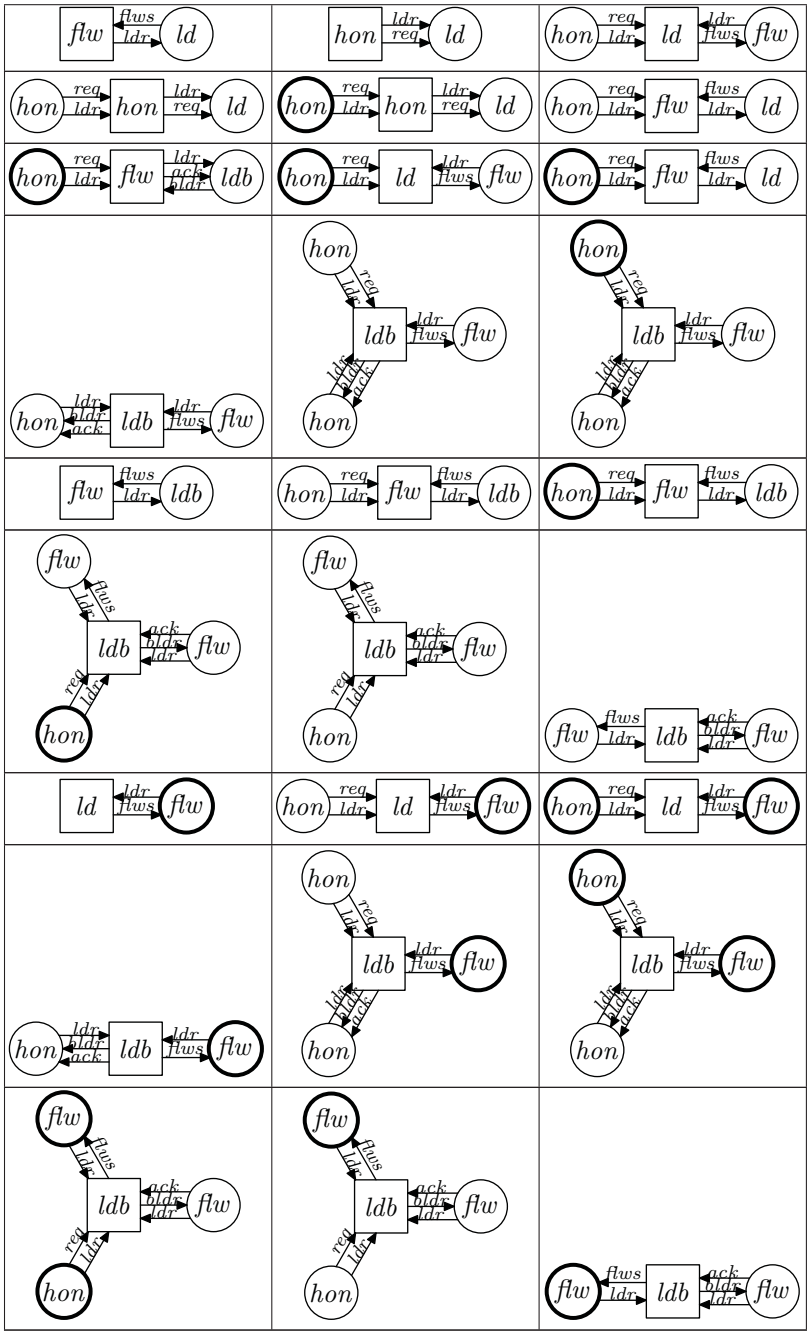
3 Results

Our tool, `astra`, expects a file in `hiralysis` format as input. The file contains a set of graph transformation rules and a start graph. We use the rules from the case study, except for the ones that deal with platoon merging and follower handover:



`astra` computes an abstraction of all graphs that can be generated by the system, resulting in the following set of stars:

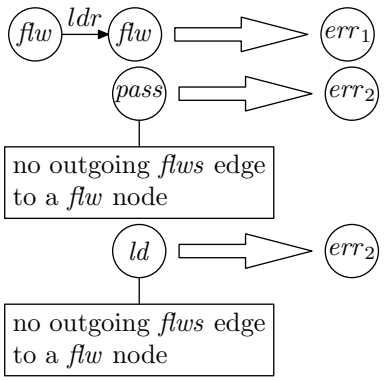



flw
flws
ldr
ld
hon
req
ldb
ack

Our tool outputs the result in graphviz, GDL, XGDL, Tulip and METAPOST format and such that it can be rendered/displayed with any tool capable of processing these formats. The graphical representation above is the METAPOST output.

4 Property evaluation

Star abstraction easily allows for evaluation of properties involving two nodes and the connections among them, such as the first and second example given in section 5.2 of the case study. This is because the stars contain each adjacent node of each node and the the edges between them. That is the only thing that the first two example properties talk about. Using the additional rules



we mark parts of the graph with error labels where the properties are violated. It is then easy to scan the result for such labels. We can verify any property that can be expressed using such rules.

Our abstraction overapproximates the set of concrete topologies that might arise. Thus, if a property of the mentioned kind (one that can be expressed by a transformation rule adding an error node) is satisfied by all topologies represented by our abstract result, we know that it is satisfied by all reachable concrete topologies as well. However, if is not satisfied in the abstract result, it might still be satisfied for the concrete result.

The computing time and memory consumption of our tool is negligible (< 1 MB, < 1 sec) on any reasonably modern machine.

5 Conclusion

Our analysis has proven powerful enough to analyze the join phase of the merge protocol in a general way, without limit to the number of processes. The resulting topologies are more complex than what can be analyzed with existing approaches, since they are not limited with respect to path length.

On the other hand, the abstraction we employed is too weak to deal with the characteristic topology structures occurring during handover. If we try to

analyze the full merge protocol, the abstraction runs into state space explosion. This is caused by the inability of the abstraction to preserve the fact that the topology has a triangular shape during handover. Accordingly, the abstraction does not preserve enough information to verify the third example property from the case study.

Since the results are still promising, we are currently implementing a new tool with an extended abstraction that is able to cope with topologies involving triangular shapes.

References

1. Peter Backes. Topology analysis of dynamic communication systems. Diploma thesis, Universität des Saarlandes, March 2008.
2. Jörg Bauer. *Analysis of Communication Topologies by Partner Abstraction*. PhD thesis, Universität des Saarlandes, 2006.
3. Jörg Bauer and Reinhard Wilhelm. Static analysis of dynamic communication systems. In *14th International Static Analysis Symposium*, 2007.

Topology Analysis of Car Platoons Merge with FujabaRT & TimedStoryCharts - a Case Study

Christian Heinzemann[1], Julian Suck[1], Ruben Jubeh[2], Albert Zündorf[2]

[1] University of Paderborn, Software Engineering Group,
Warburger Str. 100,
33098 Paderborn
chris227|jsuck@upb.de
[2] Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
ruben|zuendorf@cs.uni-kassel.de

Abstract. This paper addresses the topology analysis case study for the Transformation Tool Contest 2010. The case study presents a car platoon merge protocol with a dynamic number of participants. The task is to compute a reachability graph for all system configurations generated by the graph rewrite rules. Using the Fujaba Real-Time Tool Suite, we modeled the merge protocol as statechart in concrete syntax as this is more intuitive and use a special generator to derive the corresponding graph rewrite rules ([6, 5]). This has been combined with the hierarchical graphs library presented in [12], to compute the reachable graph transition system.

1 Introduction

This paper addresses the topology analysis case study for the Transformation Tool Contest 2010. The case study description introduces an example of a dynamic communication structure. Dynamic communication structures occur whenever an arbitrary, at design time unknown number of participants have to coordinate each other. The provided example is a group of cars that form a platoon of arbitrary length in which they travel behind a leader car. The example is structurally similar to the dynamic convoy situations (cf. [7]) in our RailCab project[3].

Systems employing dynamic communication like the car platoon often operate in safety critical environments as it is the case for both, the car platooning example and the RailCab convoys. Therefore, verification is of crucial importance for the safety of such systems to guarantee correct functionality. One inherent problem of such systems is that they often have an infinite reachable state space which makes the verification impossible without suitable abstraction or verification approaches.

We used our Mechatronic UML (e.g. [3]) approach to model the provided car platooning case study. Mechatronic UML is an adaptation of the UML allowing to model and verify systems with a dynamic communication structure ([7]). The modeling process defined by Mechatronic UML [7] allows formal verification of safety properties for arbitrary numbers of participants using inductive invariants [4]. However, inductive invariants do not support verification of all kinds of properties that are important for such systems like e.g. deadlock freedom. Since the case study explicitly requested solutions performing a reachability analysis of the protocol specification, we will show a framework for reachability analysis on graph transformation

[3] http://www.railcab.de/

systems. This enables us to verify a larger set of properties, like e.g. deadlock freedom, for a given number of instances.

We modeled the merge protocol of the car platooning case study with a statechart in concrete syntax as this is more intuitive than a direct specification in terms of graphs rewrite rules. Then, we use a partially automated generation of statecharts into our (Timed) Story Chart formalism ([6]) that allows an integrated specification of state-based protocol behavior and dynamic graph transformation. (Timed) Story Charts consist of graph transformation rules employing the behavior of the statechart and of the original graph transformation rules describing the dynamic transformations in the system structure, e.g. adding new cars to the system. We used the computation of reachability graphs as proposed in [12] to obtain the reachable graph transition system. Our framework manages sets of graphs and provides functions for copying graphs, computing hash codes for graphs, and checking isomorphisms of graphs. The resulting rules of our (Timed) Story Charts identify all possible matches for the given rule, then copy the graph and finally apply the transformation to the copy.

The concepts of the Mechatronic UML have been integrated into the Fujaba4Eclipse Real-Time Tool Suite (Fujaba RT, [10])[4] as an extension to the Fujaba4Eclipse CASE tool. The Mechatronic UML supports the specification of real-time statecharts ([2]) that allow to model timing constraints for states and transitions. We can also handle this using our Timed Story Charts and perform a reachability analysis including the timing constraints using timed story diagrams, a dialect of timed graph transformation systems ([7]). In our opinion, such timing constraints would be needed to obtain a realistic specification of the merge protocol as it is needed for the convoy coordination in our RailCab project. As this case study did not contain real-time requirements, we did not utilize the real-time capabilities of our framework.

2 Framework

This section introduces the framework which was used to model and perform the reachability analysis. Section 2.1 describes the generation of Timed Story Charts out of statecharts. Section 2.2 introduces some improvements that were made in contrast to the version described in [12].

2.1 Generating Timed Story Charts

In [6, 5], we introduced the Timed Story Chart formalism which allows to map real-time statecharts ([2]) to story diagrams extended with time. Timed Story Charts preserve the semantics of real-time statecharts while real-time statecharts can be mapped to hierarchical timed automata ([1]) which are a proper input for the Uppaal model checker ([9]) and preserve the semantics of Uppaal timed automata. Thus, Timed Story Charts have a proper semantics defined over timed automata.

The general idea is to model protocol specifications as statecharts in concrete syntax and to transform them automatically into graph rewrite rules for dynamic numbers of participants as in the car platooning case study. For a fixed number of participants, a verification using standard model checking tools is much more efficient. The first step of the transformation is to generate an object diagram for the statechart containing an object for the statechart itself as well as one object for each state of the statechart. For multiple instances of the same statechart, e.g. the statecharts of two car processes, we exploit the fact that all these instances have the same structure by generating the state structure only once. The active state of each instance is marked by an ActiveState object pointing to the object representing that state. Transitions of the statechart

[4] http://www.fujaba.de/projects/real-time.html

are mapped to Story Diagrams such that for each transition there exists one Story Diagram which executes this transition. The execution includes changing the active state, consuming received messages, generating sent messages, and performing side effects like writing local variables of the process, e.g. for storing the leader.

Asynchronous, message based communication between different statechart instances is supported by the EventQueue objects. There exists one event queue for each car process which buffers all incoming messages for the statechart. Thus, for receiving a message, a statechart reads the head message of the queue and dequeues it afterwards. For sending a message, the message is inserted into the queue of the receiving statechart by invoking the enqueue method as shown in Figure 13.

The automated transformation can generate the whole statechart structure as well as the story diagrams executing the transitions. Currently, the generation of model dependent calls like the computation of message recipients is not possible, but we are planning to extend the transformation to these aspects. The transformation itself is implemented as a model-to-model transformation using story diagrams. An example transformation rule is shown in Figure 16. It shows the top-level rule generating a class representing the transformation rule which actually executes the transition (cf. Section 2.2).

2.2 Improvements of the HierarchicalGraphsLib

The solution presented in last year's contest for the leader election protocol [12] provided a generic hierarchical graph library and a problem specific model for the leader election protocol. The graph library used runtime reflection mechanisms to provide generic copy and checkIsomorphism operations. This turned out to be inefficient, as reflective code is several times slower than generated code. We adapted the template based Fujaba code generator [8] to generate application specific copy and isomorphism check methods in the subclasses of *de.fujaba.Node* and *de.fujaba.Graph*. To be included in the isomorphism check and caching hash calculations, all associated classes have to be derived from *de.fujaba.Node* and associations have to be marked with the *usage* stereotype.

In order to ease the use of the graphs lib, we introduced a framework executing the reachability analysis which is shown in Figure 1. The core of the framework is specified in the class ReachabilityComputation. This class maintains a list of StepGraphs that were reached during the analysis and a list of StepGraphs that have to be expanded. StepGraph is a subclass of *de.fujaba.Graph* and denotes the graphs reached during the analysis. Expanding graphs, merging isomorphic graphs, and maintaining the timing computations is implemented independent from the concrete rule set. Rules are defined as subclasses of Rule. The class TransformationRule is the super class for all rules executing transitions. As we have no timing constraints in the example, the additional rules for the timing are omitted here. For specifying a reachability analysis, the user only has to specify the rule set, an initial graph, and has to instantiate the rules for the reachability analysis.

3 Modeling the merge protocol

3.1 At design time: Modeling the protocol

The underlying structure of the case study is defined by the class diagram shown in Figure 2. We introduced the class CarProcess representing the car processes. This class has four associations to itself representing the channels *leader*, *follower*, *aux*, and *bldr*. We used four unidirectional transitions in our model to imitate the behavior described in the case study. Our model would also allow to use bidirectional associations. This would enable us to express the two channels leader and follower by one 1:n association.

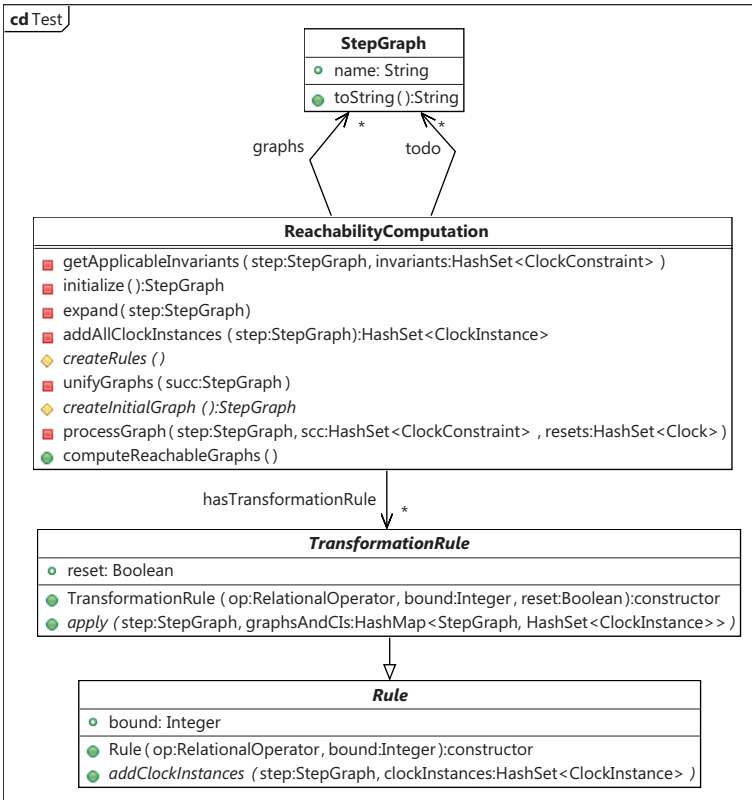


Fig. 1. Framework for the reachability analysis

The class Environment describes the environment in which the cars operate. The environment can generate new cars using the CreateNewCarProcess rule as well as it generates the car ahead (ca) messages for the cars such that they join a platoon or such that two platoons merge. There exist two additional rules for that purpose. The application of the CreateNewCarProcess rule is limited by a constant providing the desired number of cars.

The classes State, ActiveState, Parameter, EventQueue, and Statechart classes from the Timed Story Chart model described in Section 2.1 are used for the mapping of the process statechart. The Carprocess is a special parameter which can be used to attach a car process to an event. This is needed for the *ca* and *req* messages as they contain process identities. The class Car_Car_Port1 represents the statechart of the car process shown in Figure 3.

The statechart itself is directly adopted from the protocol specification contained in the case study description and uses the same states and transitions. The concrete syntax of the transitions is adopted to the syntax of our real-time statecharts. Received messages, called trigger events, are depicted in front of a "/", sent messages, called raised events, are depicted after the "/". Transition guards restrict the execution of the transition, e.g. the transition from hob to hod should only be executed if the former leader does not have any followers left. This is expressed by the guard in square brackets attached to this transition.

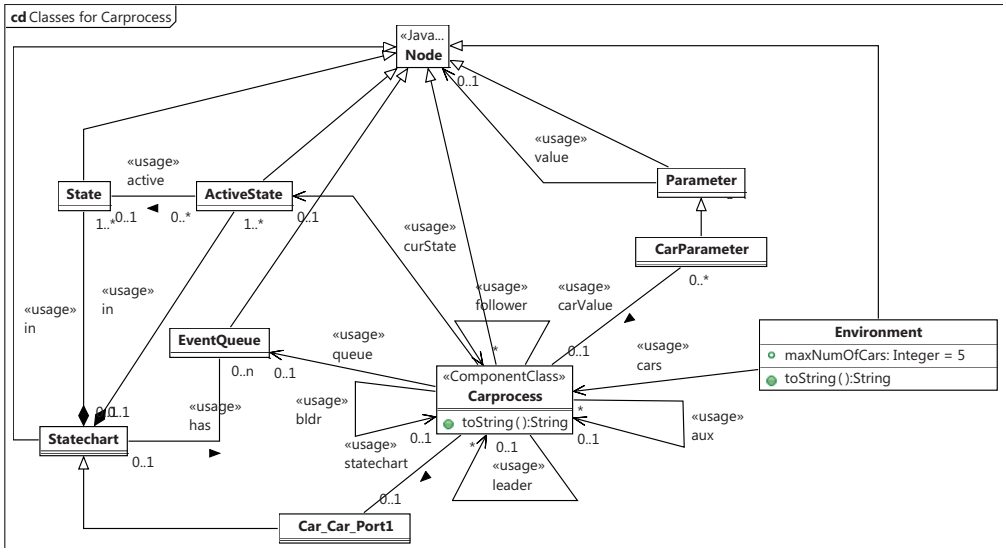


Fig. 2. Class diagram for the merge protocol specification

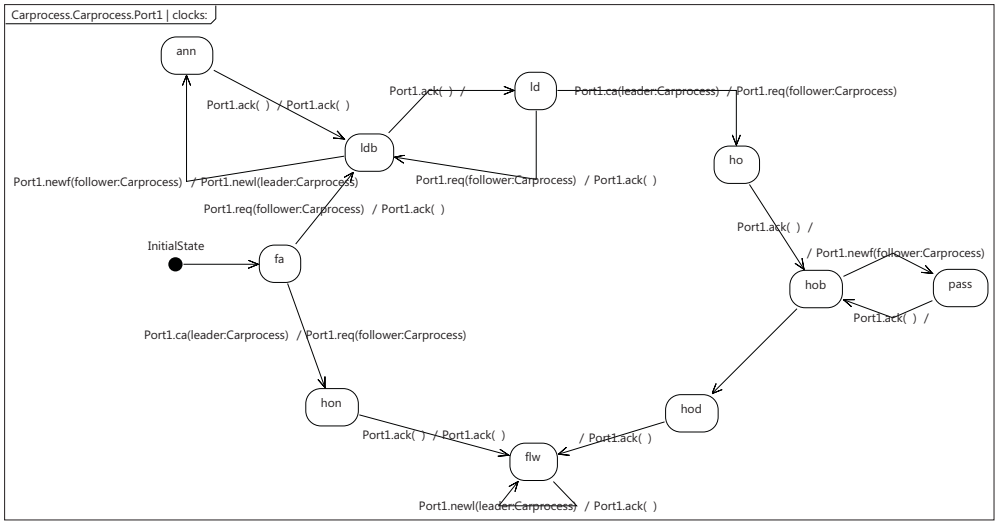


Fig. 3. Statechart specifying the protocol

3.2 At Runtime: computing the Reachability Graph

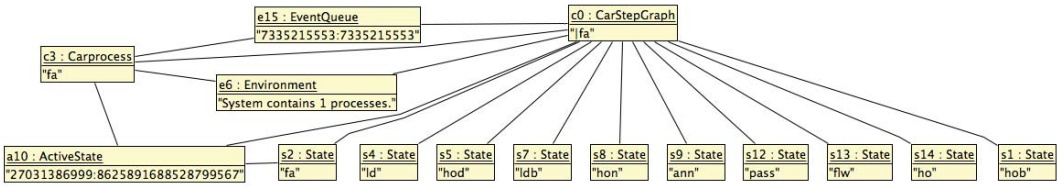


Fig. 4. Initial Graph

The runtime structure of our program is visualized by the eDOBS tool ([11]), a graphical object browser attached to the debug interface. It shows the objects in the heap, their attributes and links to other objects and helps visualizing the runtime state of our program. Figure 4 shows the initial graph we start with. There is a single car process with active state pointing to the state *fa*, an event queue, an environment and objects for each possible state a car process can be in.

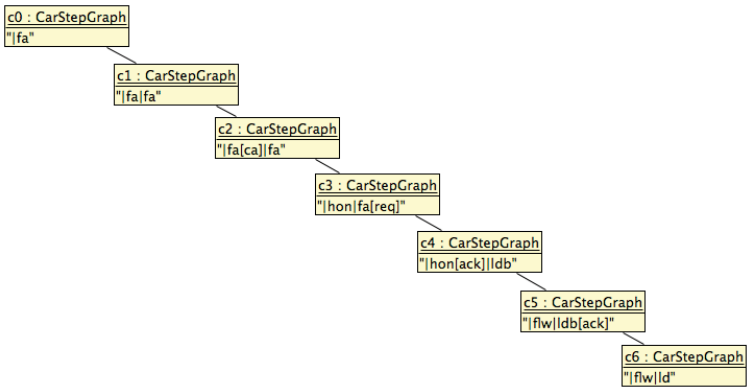


Fig. 5. Simplest reachability graph

The core of our algorithm is the `ReachabilityComputation` class. It starts with the initial graph and iteratively expands the graph by copying the graph and applying rules. After that, `unifyGraphs` (c.f. Figure 6) tries to identify isomorphic `CarStepGraphs`. Each step graph contains a certain state of the whole car process model. Figure 5 shows the structure of a reachability analysis with maximum two car processes. Each analysis state is represented as a `CarStepGraph` instance. These are ordered by an predecessor/successor-relation. In that example, each car step graph has only a single successor, as isomorphic graphs are removed after each expand step. The analysis ends with one process being the follower and the other one being the leader, which is isomorphic to *ld*|*flw*.

Figure 7 shows the full reachability graph for an analysis with max processes = 3. Isomorphic states are already removed, but the unify step also marks step graphs with same predecessor in the reachability graph.

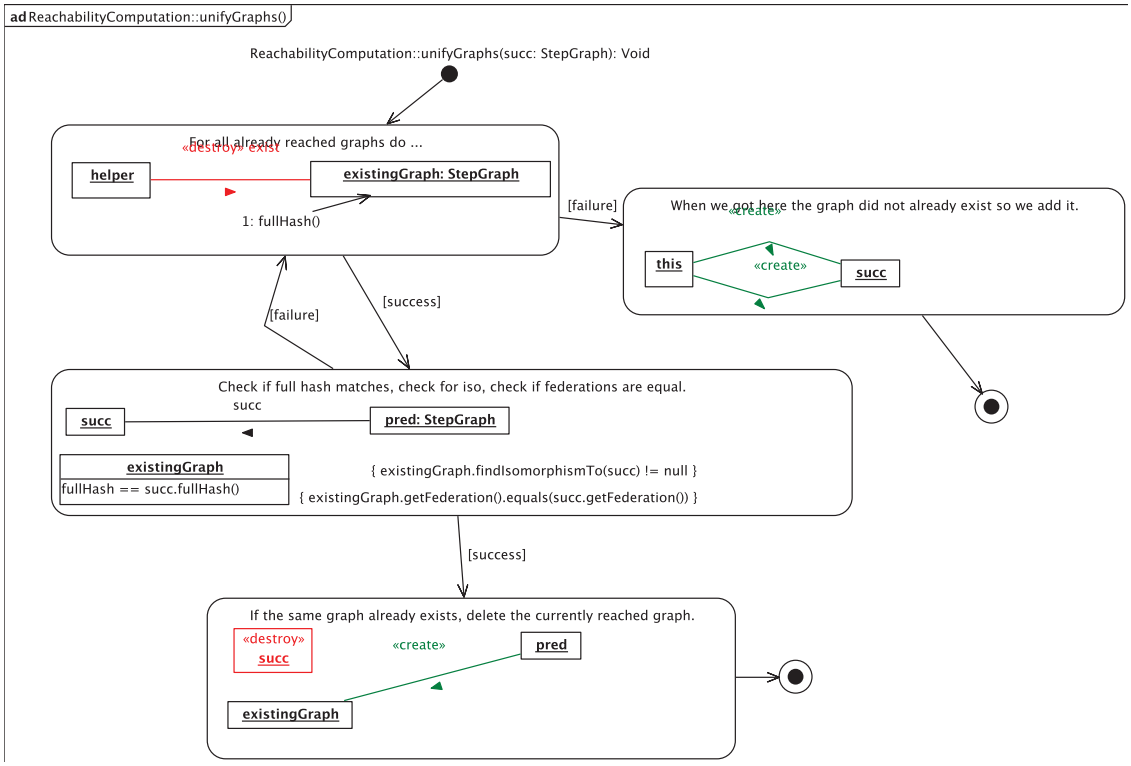


Fig. 6. unifyGraphs() rewrite rule

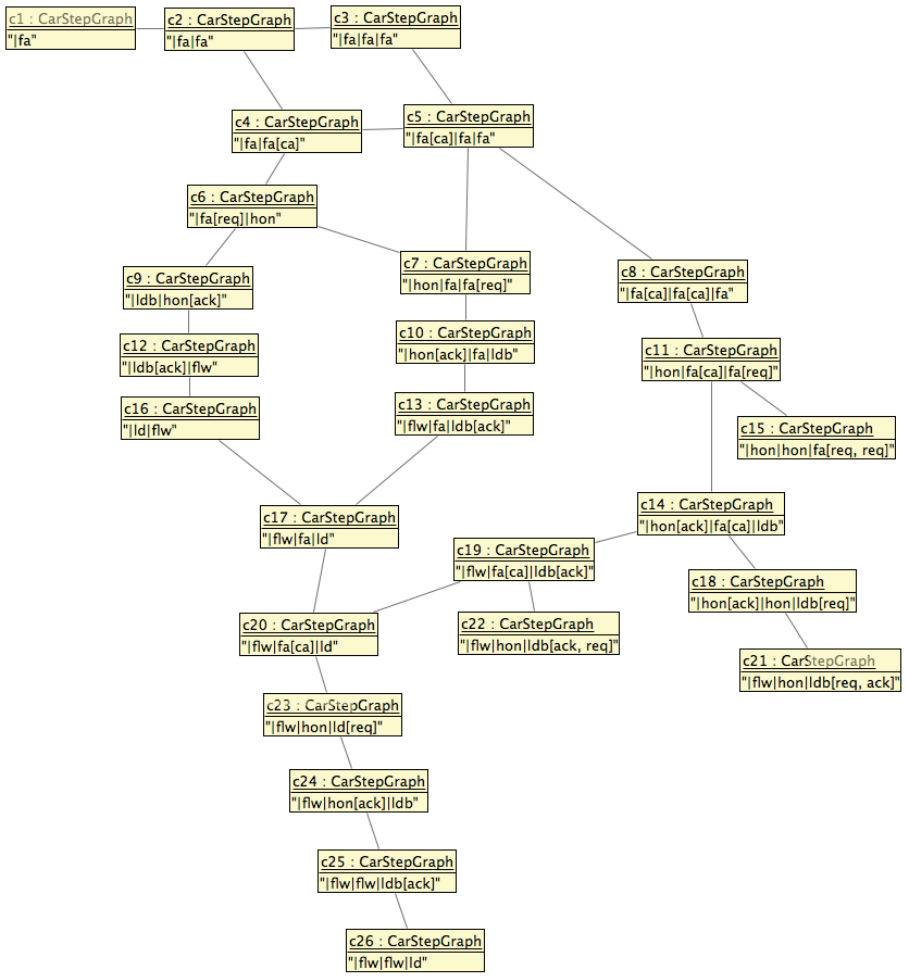


Fig. 7. Reachability graph for max processes = 3

For example, adding a new car process to the graph `c4` results in the same successor step graph as applying the rule `StartPlatoon`, which generates a *ca* environment message to `c3`, that is is the step graph `c5`.

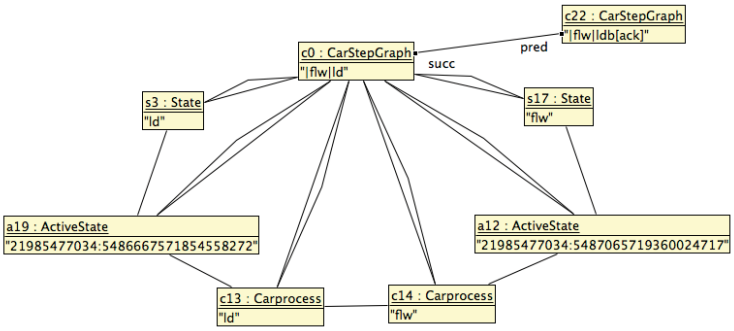


Fig. 8. Final graph for max processes = 2

The eDOBS tool can be used to verify the generated topologies. Back to the simple example with just two car processes, we inspect the final step graph, shown in figure 8. The states *ld* and *flw* associated with the car processes are correct. There is a bidirectional link between `c13` and `c14`, representing the ldr and flws edges as expected. Also, note that the previous step graph, `c22`, is the one with a process in *ldb* and an *ack* message in it's queue.

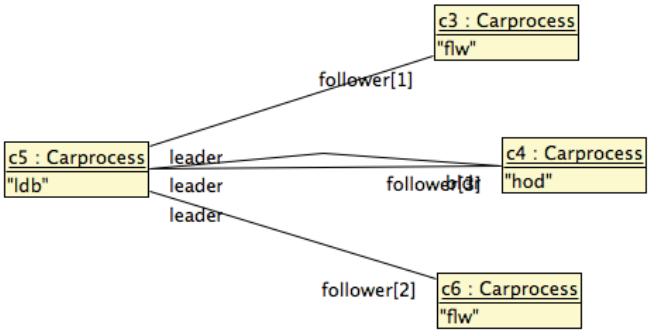


Fig. 9. Platoon Merge

More complex examples can be inspected by setting breakpoints in the corresponding rule and inspecting the step graph in eDOBS. Figure 9 shows the situation just before a platoon merge: one car process is currently in the *hod* state, just the edges between the car processes are shown. By looking at the predecessors of the current step graph, we can see previous states of the whole system.

3.3 Modeling the constraints

The given constraints to be checked during analysis are modeled as rules having a LHS only. They are derived from the `Rule` class, thus they are added to the chain of transformation rules to be checked on each iteration. They simply print out any violation. Figure 10 shows the graph pattern match rule for checking that no two processes in state *flw* are connected to each other with an *ldr* edge. Similar, figure 14 and 15 in the appendix show the two remaining constraints.

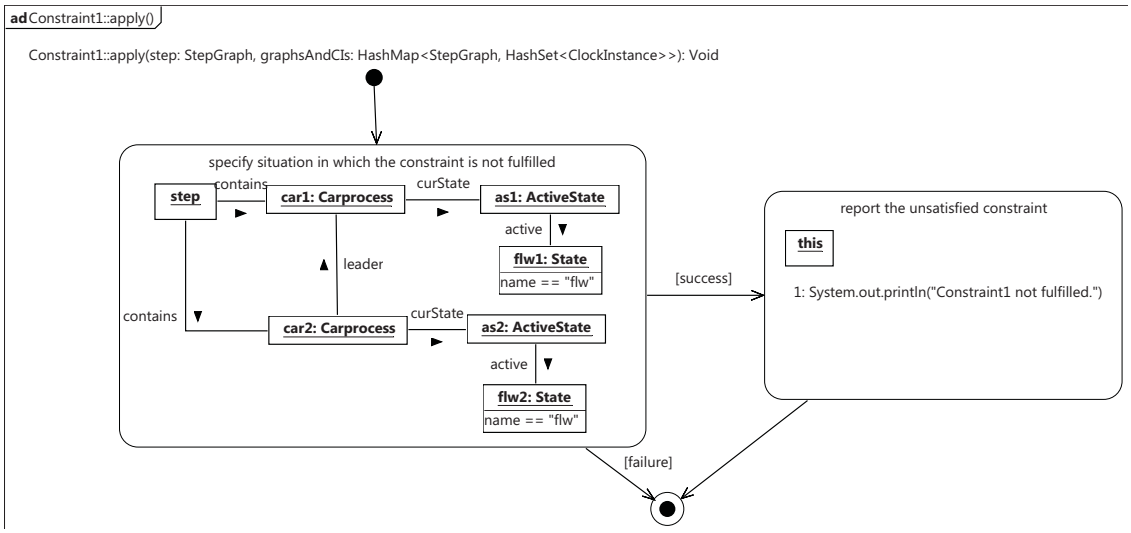


Fig. 10. Graph pattern match rule for constraint 1

Checking for deadlocks can easily performed by a post-processing searching for graphs having no successor and the reached graph transition system.

4 Evaluation

Our system can analyze topologies with up to eight car processes. Runtime measurements are presented in Table 1. Each line represents a full analysis with the maximum number of car processes limited to `max processes`. Column two shows the number of non-isomorphic graphs resulting for that analysis run and

column three shows the time needed for the computation. Column four denotes the number of nodes per graph while Column five shows the memory consumption of the computation.

max processes	graphs	Runtime (s)	graphsize	memory
2	7	0	21	?
3	26	0	26	?
4	96	1	31	7MB
5	348	2	36	42MB
6	1317	12	41	163MB
7	5100	347	46	733MB
8	20353	14046	51	5.8GB

Table 1. Evaluation results

Figure 11 shows the nodes per graph. This grows linear as expected. In contrast to that, our analysis runtime grows exponentially, as shown in figure 12. Analyzing the system with up to eight processes already requires almost 4 hours runtime, so calculating nine seems to be impossible with the current approach, as well as with the current memory requirements. The graph isomorphism check still seems to be inefficient when many isomorphisms are expected.

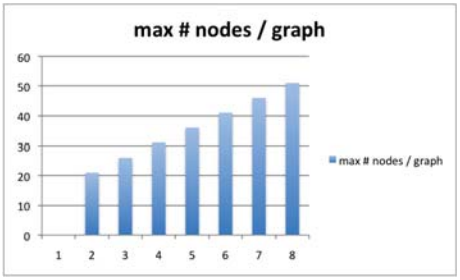


Fig. 11. Number of nodes per graph

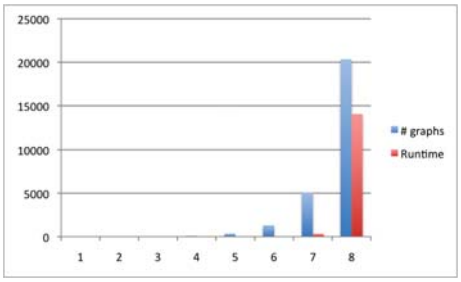


Fig. 12. Problem size versus runtime (in seconds)

5 Conclusions

We have shown an approach to model the topology analysis case study by using concrete statechart syntax. The statecharts are automatically transformed into Story Diagrams and executed by a framework computing the set of reachable graphs. Our evaluation results have shown, that checking more than 8 car processes is currently not possible using our tool regarding time and memory consumption for the computations with 8 cars. Although not requested in this case study, our framework is able to handle timing constraints which would lead to a more realistic specification of the merge protocol.

In FujabaRT, we are now able to use a combination of real-time statecharts and real-time transformation rules to specify the behavior of a system. Using our generators and the reachability graph framework, we

are able to compute the reachability graph for the specified system. In addition, one may add constraint checking rules to the reachability graph framework.

Currently, we use normal rules to specify our constraints that check for the existence or absence of certain invariant structures. This enables us to check CTL formulas of the kind $EF\varphi$ and $\neg AG\varphi$ for some graph invariant φ. We plan to extend our checking approach to a greater class of CTL formulas in future work. It is also possible to extend our approach by using inductive invariants to verify properties for infinite state systems ([4]).

The performance of our reachability analysis has been improved since last year. However, our framework introduces a certain overhead due to the unused timing capabilities which require some additional search operations in the graph. Additionally, our current Timed Story Chart approach uses one object for each state of the statechart which increases the number of objects per graph by a constant and we plan to investigate whether a more compact statechart encoding yields better results. Finally, this case study clearly identified the isomorphism checks of our graph lib framework as a bottleneck and we have developed a number of ideas for further improvements for the next year.

References

1. A. David, M. O. Möller, and W. Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 208–241. Springer Berlin / Heidelberg, 2002.
2. H. Giese and S. Burmester. Real-time statechart semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany, June 2003.
3. H. Giese, S. Henkler, M. Hirsch, V. Roubin, and M. Tichy. Modeling techniques for software-intensive systems. In D. P. F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*, pages 21–58. Langston University, OK, 2008.
4. H. Giese and D. Schilling. Towards the Automatic Verification of Inductive Invariants for Infinite State UML Models. Technical Report tr-ri-04-252, University of Paderborn, Paderborn, Germany, December 2004.
5. C. Heinzemann, S. Henkler, and M. Hirsch. Refinement checking of self-adaptive embedded component architectures. Technical Report tr-ri-10-313, University of Paderborn, 2010.
6. C. Heinzemann, S. Henkler, and A. Zündorf. Specification and refinement checking of dynamic systems. In P. V. Gorp, editor, *Proceedings of the 7th International Fujaba Days*, pages 6–10, Eindhoven University of Technology, The Netherlands, November 2009.
7. S. Henkler, M. Hirsch, C. Priesterjahn, and W. Schäfer. Modeling and verifying dynamic communication structures based on graph transformations. In *Proc. of the Software Engineering 2010 Conference, Paderborn, Germany, 22.-26.2.2010*, 2010. accepted.
8. L. Geiger, C. Schneider, C. Record. Template- and modelbased code generation for MDA-Tools. *3rd International Fujaba Days 2005, Paderborn, Germany*, September 2005.
9. K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, Oct. 1997.
10. C. Priesterjahn, M. Tichy, S. Henkler, M. Hirsch, and W. Schäfer. Fujaba4eclipse real-time tool suite. In *Model-Based Engineering of Embedded Real-Time Systems (MBEERTS)*, LNCS, pages 1–7. Springer, 2009. accepted.
11. The EDobs Dynamic Object Browser. http://www.se.eecs.uni-kassel.de/typo3/index.php?edobs, 2006.
12. A. Zündorf. Model Checking the Leader Election Protocol with Fujaba. In *GraBaTs 2009, 5th International Workshop on Graph-Based Tools*, Zurich, Switzerland, 2009.

A Appendix

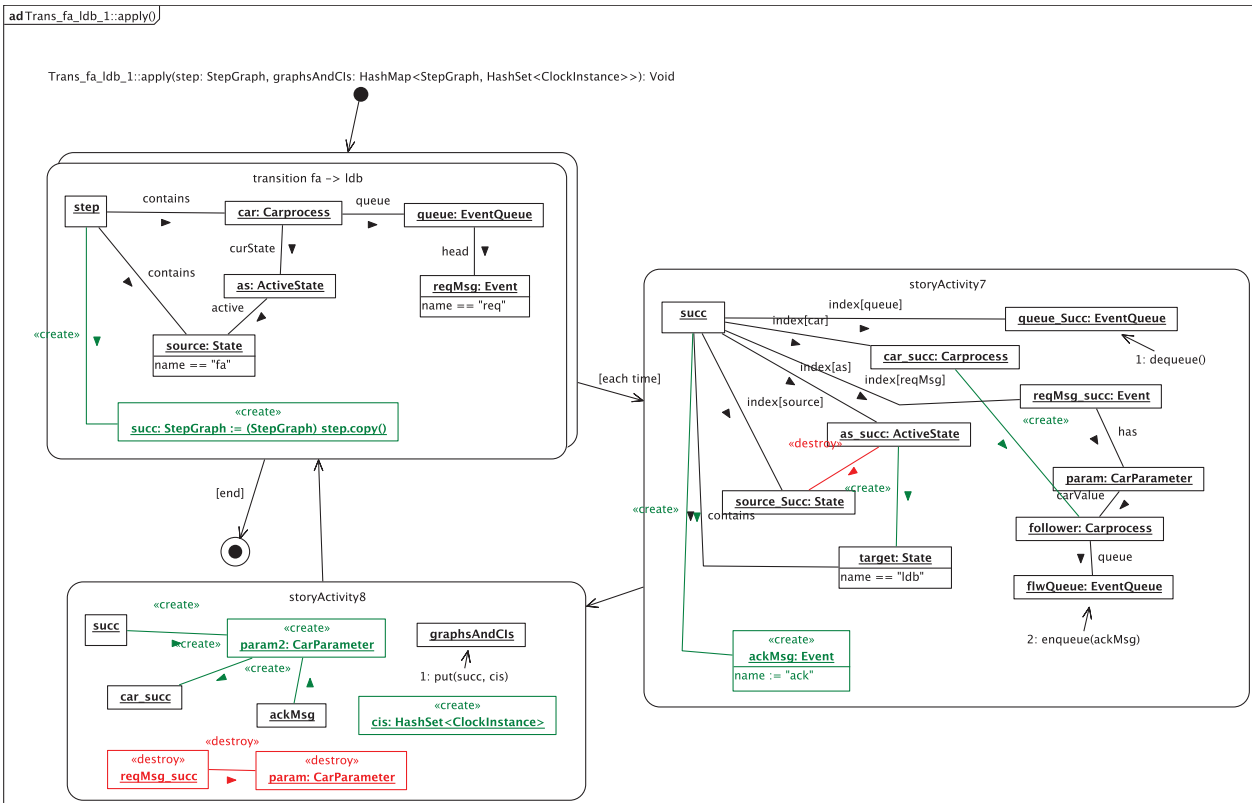


Fig. 13. Example transformation rule rule: fa to ldb

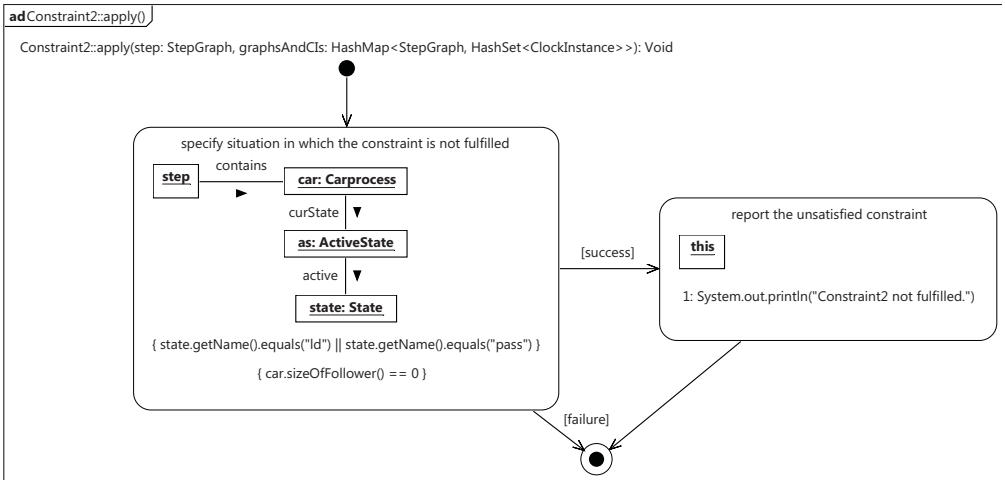


Fig. 14. Graph pattern match rule for constraint 2

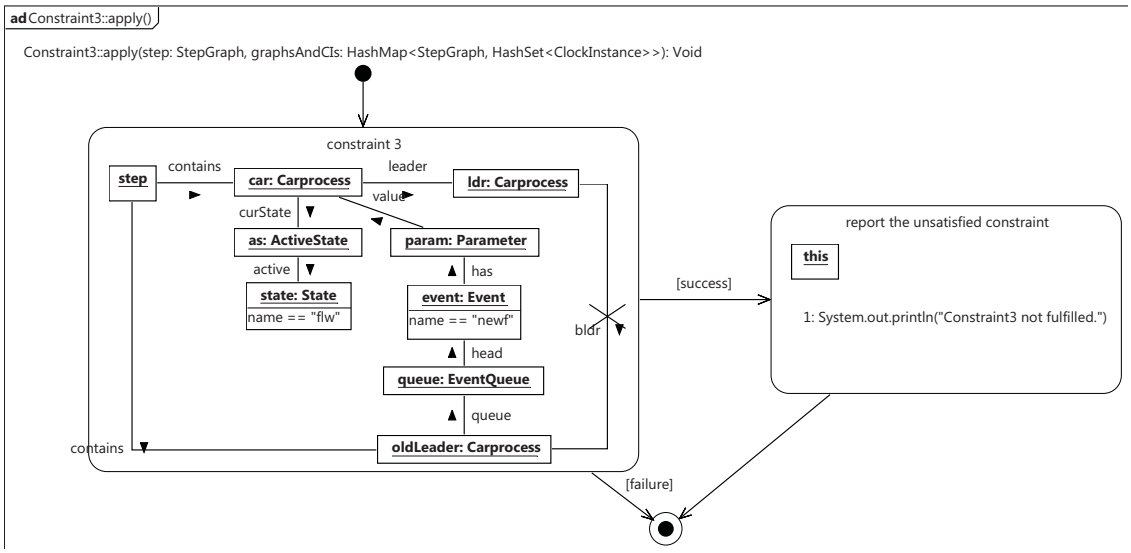


Fig. 15. Graph pattern match rule for constraint 3

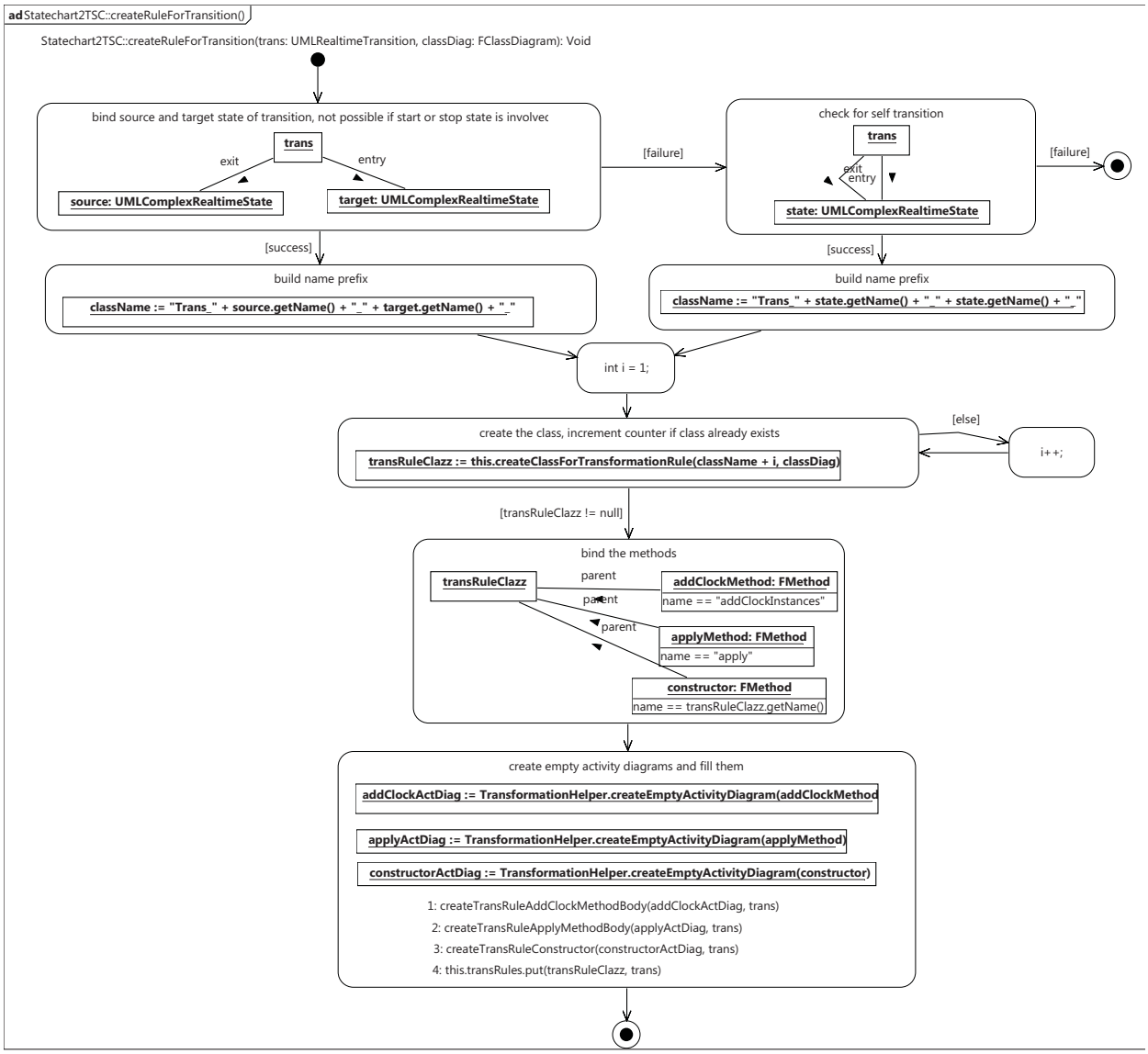


Fig. 16. Top-level Story Diagram translating a transition into a Story Diagram rule

Ecore to GenModel Case Study

Dimitrios S. Kolovos[1], Louis M. Rose[1],
Richard F. Paige[1], Juan De Lara[2]

[1] Department of Computer Science,
University of York, Y010 5DD, York, UK,
{dkolovos,louis,paige}@cs.york.ac.uk
[2] Universidad Autónoma de Madrid,
Juan.deLara@uam.es

1 Introduction

Ecore is the metamodelling language of the Eclipse Modeling Framework (EMF)[1], which is arguably the most widely-used modelling framework today. EMF provides two ways for instantiating models that conform to an Ecore metamodel: by reflection or by code generation. Of interest to our case study is the latter. EMF provides a toolkit which can generate a set of Java classes as well as a tree-based editor from an Ecore metamodel. This process is performed in two steps. In the first step, the Ecore metamodel is transformed into a GenModel model which can accommodate for additional implementation-specific information. Such information includes the name of the Java package under which the generated code will be placed, as well as presentation and persistence-related options. Then a JET-based model-to-text transformation consumes the GenModel in order to generate the functional Java code.

2 Proposed Case Study

Currently, the transformation that generates a GenModel model from an Ecore metamodel is performed using Java. In this case study we propose using a model-to-model transformation language for reimplementing the same transformation. This transformation has an interesting property: generated elements in the GenModel refer back to elements in the original Ecore metamodel. For instance, when a GenClass is generated from an EClass, the GenClass refers to the original EClass through its ecoreClass property.

[1] `http://www.eclipse.org/emf`

3 Extension 1

As stated above, once the GenModel is generated, users can modify it in order to configure low-level aspects of the resulting Java implementation. If the Ecore model then changes, the built-in GenModel editor provides a reconciliation facility that can - in most cases - propagate changes from the Ecore to the GenModel model while preserving any user-set attribute values. There are at least two shortcomings in this approach. First, by inspecting a GenModel model it is challenging to distinguish between user-set attributes and default/generated ones. Also, in case of major changes in the Ecore metamodel, the reconciler can fail and therefore the user needs to re-generate and re-customize the GenModel from scratch.

To overcome these limitations, we propose specifying GenModel options as annotations in the source Ecore metamodel itself and then populating these options through the Ecore2GenModel transformation [2]. For instance, consider the following metamodel (in Emfatic[3])

```
@namespace(uri="flowchartdsl", prefix="")
@emf.gen(basePackage="org.ttc10")
package flowchartdsl;

class Flowchart {
val Node[*] nodes;
val Transition[*] transitions;
}

abstract class Node {
attr String name;
}

class Transition {
   attr String name;
   ref Node source;
   ref Node target;
}

class Action extends Node {

}

class Decision extends Node {

}
```

[2] An alternative would be to use a separate model to specify these annotations but we have chosen the embedded annotations approach primarily for usability reasons.

[3] http://www.eclipse.org/gmt/epsilon/doc/articles/emfatic/

With the proposed extension, the value of the emf.gen basePackage annotation on the root EPackage (flowchartdsl) would be set as the value of the basePackage attribute of the generated flowchartdsl GenPackage.

4 Extension 2

The simplest way of propagating annotation values from the Ecore over to the respective GenModel elements is to do this explicitly for each annotation/attribute pair. However, if the transformation language supports reflection, this can also be performed by a generic function reflectively as follows (in pseudocode):

```
function copyAnnotations(Ecore element, GenModel element) {
   for (annotation in Ecore element annotations) {
      if (GenModel element has an attribute called annotation.name) {
         if (the type of the attribute is compatible
                  with the value of the annotation) {

            do the conversion and assign the value;
         }
         else {
            report a warning;
         }
      }
      else {
         ignore the annotation;
      }
   }
}
```

5 Rationale

This transformation tests two very useful features of a transformation language: support for establishing cross-model references and support for reflection.

6 Availability of metamodels and sample models

Both the Ecore and the GenModel metamodels are built in EMF. Ecore models can be constructed with the graphical Ecore diagram editor, the built-in tree-based Ecore editor, the Emfatic textual syntax or otherwise. GenModels can then be derived from them using the respective EMF wizard.

7 Implementation with ETL

The proposed transformation has been recently implemented with the Epsilon Transformation Language (ETL)[4] and is available here:

`http://bit.ly/bmWbWQ`

A screencast demonstrating the transformation in action is also available here:

`http://bit.ly/cusuKd`

[4] `http://www.eclipse.org/gmt/epsilon/doc/etl`

Modeling the "Ecore to GenModel" Transformation with EMF Henshin

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

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

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

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

1 Introduction: Transforming Ecore to GenModel

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

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

2 Transformation Concepts of Henshin

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

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

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

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

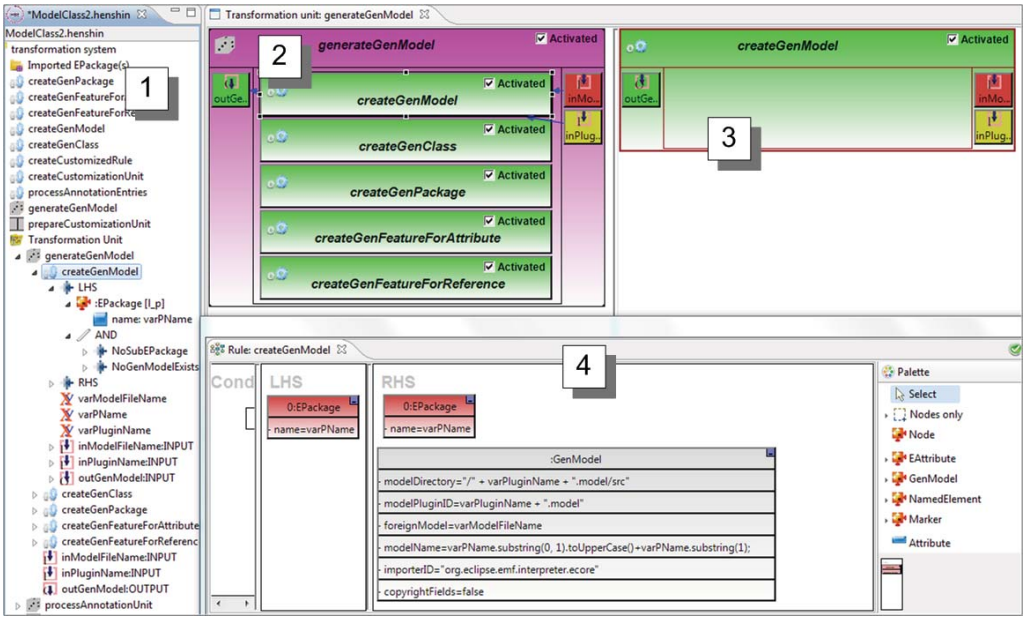


Fig. 1. Henshin GUI with tree view (1), transformation unit editor (2) and (3), and rule editor (4).

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

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

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

3 The Ecore2GenModel Transformation

Fig. 2 shows our self-defined mapping model used to connect a source EMF model *Ecore* with a target EMF model *GenModel*. In detail, in the center of Fig. 2 class `Marker` is depicted, whose instances mark annotation entries (`EStringToStringMapEntry`) of the source model that are processed during the transformation. Instances of class `Rel` keep track of Ecore objects (`EModelElement`) and their corresponding *GenModel* objects (`GenBase`).

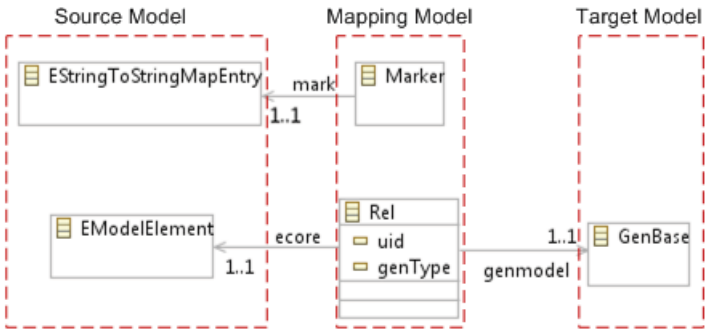


Fig. 2. Mapping model for the *Ecore2Genmodel* transformation and its connections to the source and target meta models

Please note that we actually do not need any mapping model at all for the basic *Ecore2GenModel* transformation since the *GenModel* model already contains references to the *Ecore* model. However, here we already prepare our solution of Extension 2 (see Section 4) by creating additional helper structures for translating also *annotated* EMF models.

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

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

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

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

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

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

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

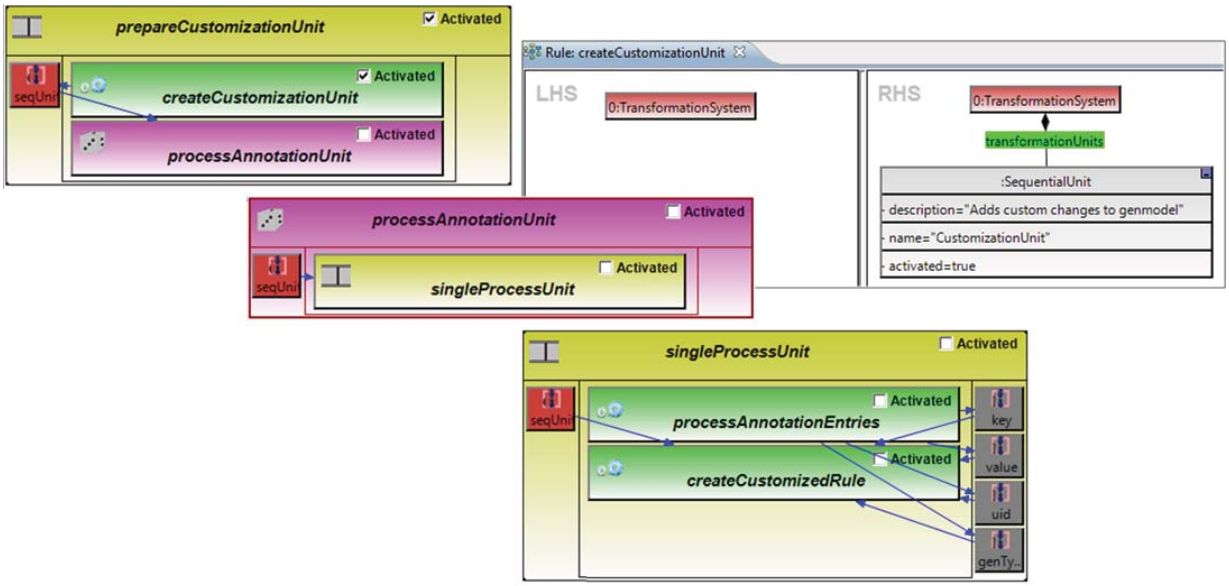


Fig. 3. Transformation units for processing annotated Ecore models

5 Conclusion

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

References

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

A Java Code of the Transformation Application

```
1
2  package tcc10;
3
4  import java.io.File;
5  import java.io.IOException;
6
7  import org.eclipse.emf.codegen.ecore.genmodel.GenModel;
8  import org.eclipse.emf.codegen.ecore.genmodel.GenModelPackage;
9  import org.eclipse.emf.codegen.ecore.genmodel.impl.GenModelPackageImpl;
10 import org.eclipse.emf.common.util.URI;
11 import org.eclipse.emf.ecore.EObject;
12 import org.eclipse.emf.ecore.EPackage;
13 import org.eclipse.emf.ecore.resource.Resource;
14 import org.eclipse.emf.ecore.resource.ResourceSet;
15 import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
16 import org.eclipse.emf.ecore.xmi.impl.EcoreResourceFactoryImpl;
17 import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;
18 import org.eclipse.emf.henshin.common.util.EmfGraph;
19 import org.eclipse.emf.henshin.interpreter.EmfEngine;
20 import org.eclipse.emf.henshin.interpreter.UnitApplication;
21 import org.eclipse.emf.henshin.model.SequentialUnit;
22 import org.eclipse.emf.henshin.model.TransformationSystem;
23 import org.eclipse.emf.henshin.model.TransformationUnit;
24 import org.eclipse.emf.henshin.model.impl.HenshinPackageImpl;
25 import org.eclipse.emf.henshin.model.resource.HenshinResourceFactory;
26
27 /**
28  * This implementation of an Ecore to Genmodel transformation by <a
29  * href="http://www.eclipse.org/modeling/emft/henshin/">Henshin</a> was
      created
30  * along the <a
31  * href="http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/">Transformation
      Tool
32  * Contest 2010</a> organized as satellite workshop to <a
33  * href="http://malaga2010.lcc.uma.es/">TOOLS 2010</a>.<br>
34  * Authors are (in alphabetical order):
35  * <ul>
36  * <li>Enrico Biermann
37  * <li>Claudia Ermel
38  * <li>Stefan Jurack
39  * </ul>
40  *
41  * <i>Remark:</i> As proof of concept only, in the following source
      (.ecore) and
42  * target (.gemodel) model files are hard-coded. However, an adaption to a
43  * full-fledged plugin providing a context menu entry for ecore files is
44  * straightforward.
45  */
46 public class Ecore2GenmodelTrafo {
```

```
47
48          /** Definition of a number of file paths */
49          private static final String BASE = "model/";
50
51          /** Mapping model */
52          private static final String ECORE_E2G = "ecore2gen.ecore";
53          private static final String ECORE_E2G_FULL = BASE + ECORE_E2G;
54          /** Henshin file containing relevant rules */
55          private static final String HENSHIN_E2G_FULL = BASE
56                          + "Ecore2Genmodel.henshin";
57          /** Ecore source model to be transformed */
58          private static final String ECORE_SOURCE = "flowchartdsl.ecore";
59          private static final String ECORE_SOURCE_FULL = BASE + ECORE_SOURCE;
60          /** Genmodel target model */
61          private static final String GENMODEL_TARGET_FULL = BASE
62                          + "flowchartdsl2.genmodel";
63
64          /** Common resource set */
65          ResourceSet resourceSet = new ResourceSetImpl();
66
67          /**
68           * Method comprising the main control flow for the transformation.
69           */
70          public void generateEcore2Genmodel() {
71
72                  initializeResourceFactories();
73
74                  TransformationSystem ts = (TransformationSystem)
                        loadModel(HENSHIN_E2G_FULL);
75                  EPackage mappingModel = (EPackage)
                        loadModel(ECORE_E2G_FULL);
76
77                  EPackage ecoreModel = (EPackage)
                        loadModel(ECORE_SOURCE_FULL);
78
79                  // Create Henshin interpreter objects
80                  EmfGraph graphM = new EmfGraph();
81                  graphM.addRoot(ecoreModel);
82                  EmfEngine engineM = new EmfEngine(graphM);
83
84                  // Generate genmodel from ecore model (without annotations).
85                  TransformationUnit unit1 =
                        ts.findUnitByName("generateGenModel", true);
86                  UnitApplication unitApp1 = new UnitApplication(engineM,
                        unit1);
87                  // file name and plugin name cannot be reliably deduced by
                        the model
88                  // elements thus need to be set.
89                  unitApp1.setPortValue("inModelFileName", ECORE_SOURCE);
90                  unitApp1.setPortValue("inPluginName", ecoreModel.getName());
91                  boolean result = unitApp1.execute();
```

```
92
93                  graphM.addRoot(ts);
94                  graphM.addRoot(GenModelPackage.eINSTANCE);
95                  graphM.addRoot(mappingModel);
96
97                  // Process annotations and generate related Henshin rules.
98                  TransformationUnit unit2 = ts.findUnitByName(
99                                  "prepareCustomizationUnit", true);
100                 UnitApplication unitApp2 = new UnitApplication(engineM,
                        unit2);
101                 unitApp2.execute();
102
103                 // Apply generated rules to transfer annotations to the
                        genmodel.
104                 SequentialUnit customizationUnit = (SequentialUnit) unitApp2
105                                 .getPortValue("seqUnit");
106                 UnitApplication unitApp3 = new UnitApplication(engineM,
107                                 customizationUnit);
108                 unitApp3.execute();
109
110                 // Save resulting genmodel.
111                 if (result) {
112                         System.out.println("Successful");
113                         GenModel gm = (GenModel)
                                unitApp1.getPortValue("outGenModel");
114                         saveGenModel(gm);
115                 } else {
116                         System.out.println("Not successful");
117                 }// if else
118
119         }// generateEcore2Genmodel
120
121         /**
122          * Saves the content of the genmodel to the specified file (see
123          * {@link #createGenModelResource()}).
124          *
125          * @param gen
126          */
127         private void saveGenModel(GenModel gen) {
128                 URI modelUri = URI.createFileURI(new
                        File(GENMODEL_TARGET_FULL)
129                                 .getAbsolutePath());
130                 Resource res = resourceSet.createResource(modelUri,
                        "genmodel");
131                 try {
132                         res.getContents().add(gen);
133                         res.save(null);
134                 } catch (IOException e) {
135                         e.printStackTrace();
136                 }// try catch
137         }// saveGenModel
```

```
138
139         /**
140          * Loads the model at the given path and returns the root element.
141          *
142          * @param modelPath
143          * @return
144          */
145         private EObject loadModel(String modelPath) {
146                 URI modelUri = URI.createFileURI(new
                        File(modelPath).getAbsolutePath());
147                 Resource resourceModel = resourceSet.getResource(modelUri,
                        true);
148                 return resourceModel.getContents().get(0);
149         }// loadEmfModel
150
151         /**
152          * Registers appropriate resource factories for <b>ecore</b>,
153          * <b>genmodel</b> and <b>henshin</b> files.
154          */
155         private void initializeResourceFactories() {
156                 Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
157                                 "ecore", new EcoreResourceFactoryImpl());
158                 Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
159                                 "genmodel", new XMIResourceFactoryImpl());
160                 Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
161                                 "henshin", new HenshinResourceFactory());
162
163                 // Initialize packages
164                 GenModelPackageImpl.init();
165                 HenshinPackageImpl.init();
166         }// initializeResourceFactories
167
168         /**
169          * @param args
170          */
171         public static void main(String[] args) {
172                 Ecore2GenmodelTrafo s = new Ecore2GenmodelTrafo();
173                 s.generateEcore2Genmodel();
174         }// main
175
176 }// class
```

B Rules contained in Unit `generateGenModel`

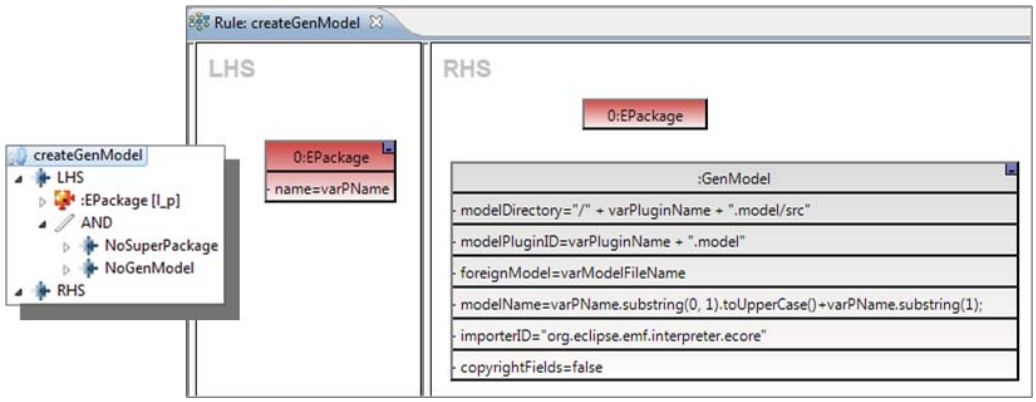


Fig. 4. Rule `createGenModel`

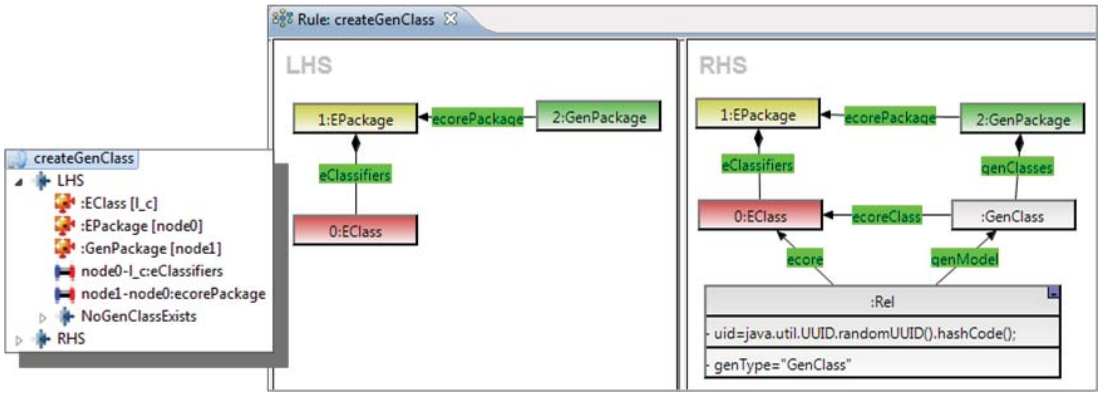


Fig. 5. Rule `createGenClass`

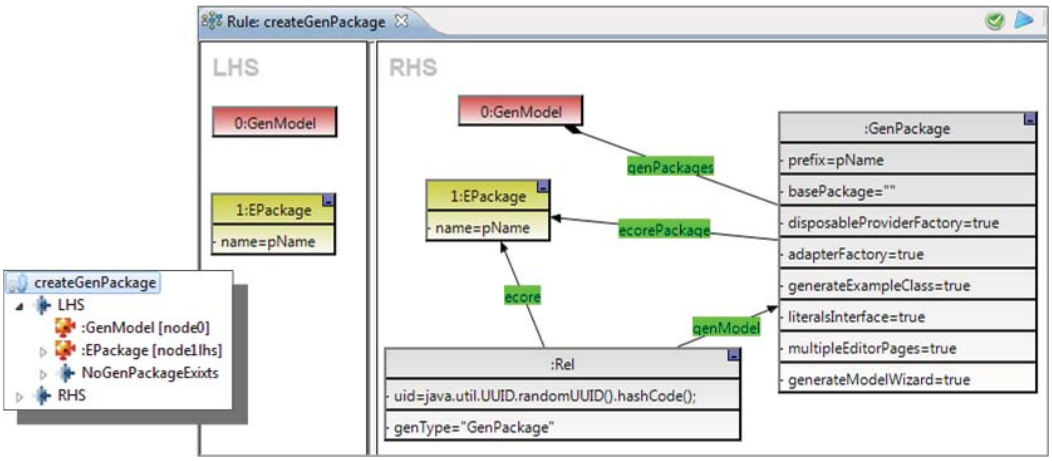


Fig. 6. Rule `createGenPackage`

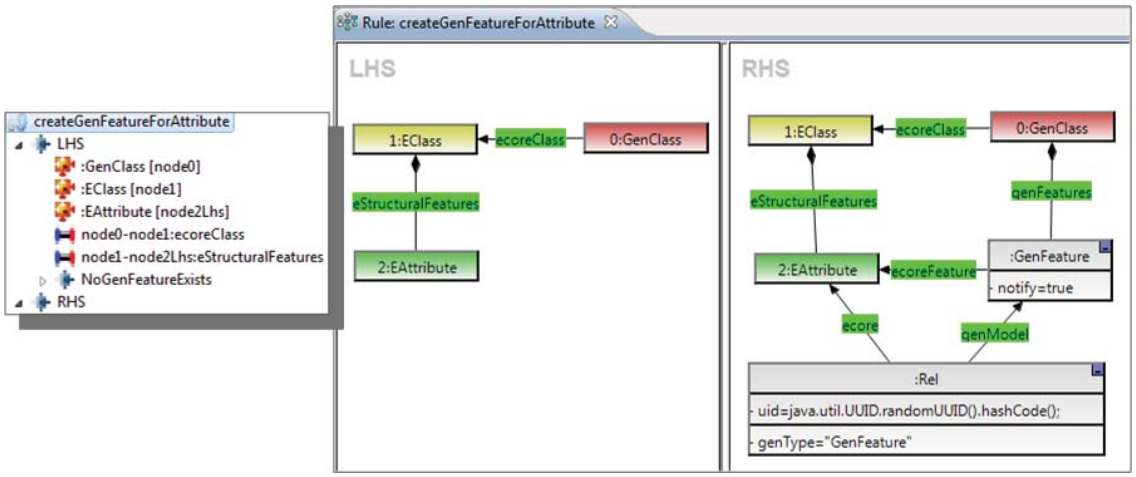


Fig. 7. Rule `createGenFeatureForAttribute`

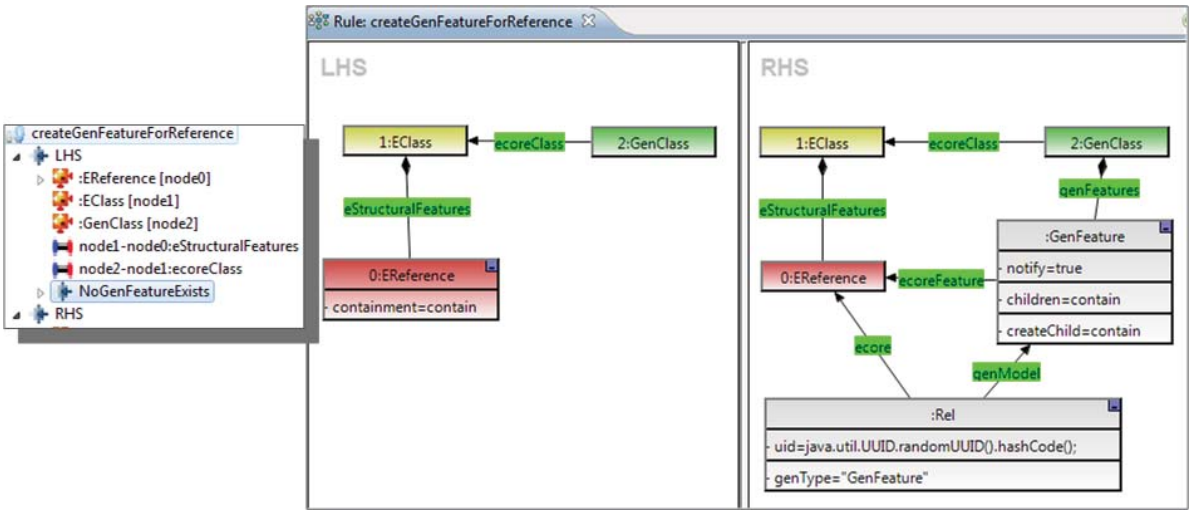


Fig. 8. Rule `createGenFeatureForReference`

C Rules contained in Unit `singleProcessUnit`

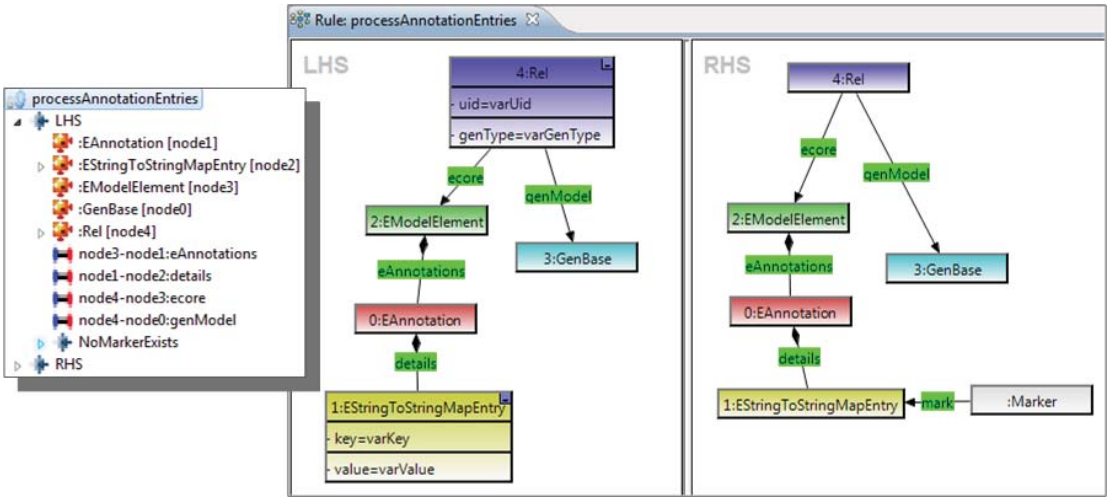


Fig. 9. Rule `processAnnotationEntries`

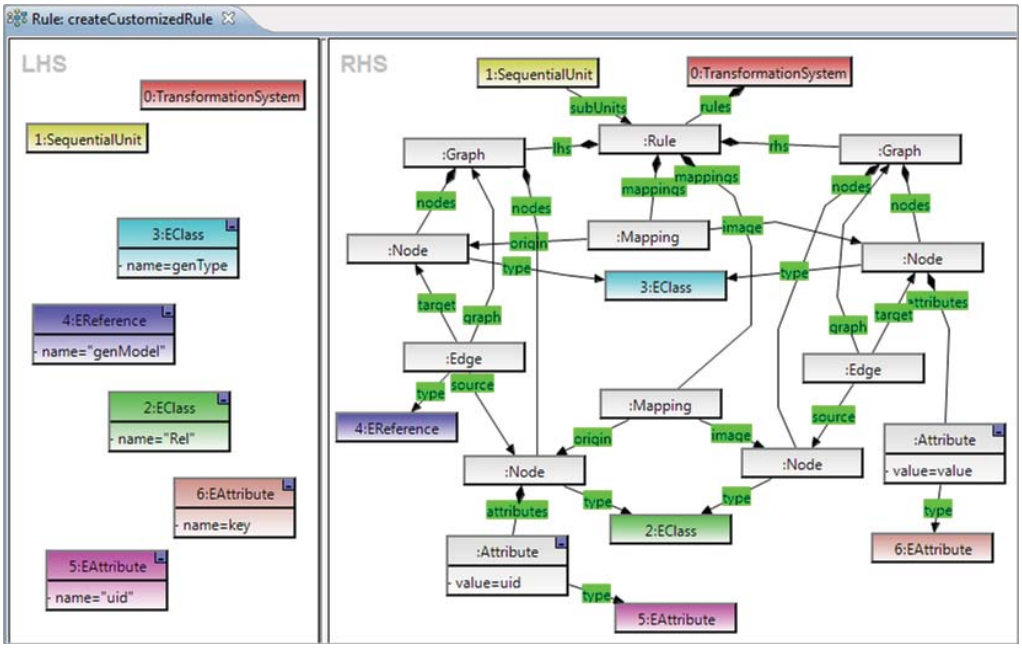
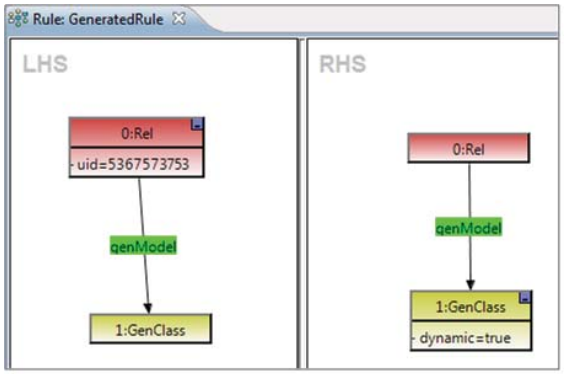


Fig. 10. Rule `createCustomizedRule`



generated by rule createCustomizedRule with the parameters:

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

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

Ecore2GenModel with Mitra and GEF3D

Jens von Pilgrim
Jens.vonPilgrim@FernUni-Hagen.de
FernUniversität in Hagen

The case study proposed by Kolovos et al.[1] describes the transformation of an Ecore model to a GenModel. In order to populate the features of the GenModel elements, Kolovos et all propose to use annotations. The solution described here exactly implements that technique using a transformation language called *Mitra*. Besides, an alternative mechanism is implemented, using generic model markers, which are stored in a separate model.

1. Mitra and GEF3D

The language used in this solution is called Mitra, a short version of micro transformation. Mitra is an imperative language optimized for computer-assisted transformation (CAT), that is a transformation which rules are selectively invoked by the user but executed by a transformation engine.[2] Since the user is highly involved in the transformation process, the user interface is important. Many models used in Model Driven Development (MDD) are visualized with a graphical syntax, e.g., UML or Ecore diagrams. Furthermore, model-to-model (M2M) transformations often operate on multiple models, e.g., a source and a target model. Thus, visualizing a single diagram is not sufficient, instead multiple diagrams are to be displayed. For that reason, Mitra is combined with GEF3D[3], a framework enabling 3D diagram editors. Existing 2D editors can easily be adapted, called 3D-fied, in order to be used in 3D: the 2D output of these editors is projected onto a plane in the 3D scene.[4]

Two different user interface concepts are applied, depending on the parameters of the transformation rule The two concepts are illustrated in Figure 1 and Figure 2.

The first concept is called *dropformation*, a short version of drag-and-drop-and-transform. One rule provided by the solution transforms an Ecore class to an UML:

```
manual traced EClass2UMLClass
    (from Ecore.EClass s, into UML.Package p) : (create UML.Class t)
```

The user interface can exploit the parameter modifiers (**from**, **into**) in order to determine the arguments provided by the user. From the user's point of view, the transformation looks like a simple drag-and-drop operation. The dragged Ecore class (the **from** parameter) is simply dropped onto the UML package (the **into** parameter), and the new UML class is created by the transformation.

[1] Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Juan de Lara. Ecore to genmodel case study. In *Transformation Tool Contest 2010*, 2010

[2] The syntax of Mitra is quite similar to Java, except local variables are to be declared with the keyword **var**. In contrast to Java, multiple return parameters can be declared, however this feature is not used in the example here. Other special features are explained in the following.

[3] `http://www.eclipse.org/gef3d`

[4] In the solution, 3D-fied versions of the Ecore Tools editor and UML2 Tools editors are used. These versions are part of GEF3D's example plugin.

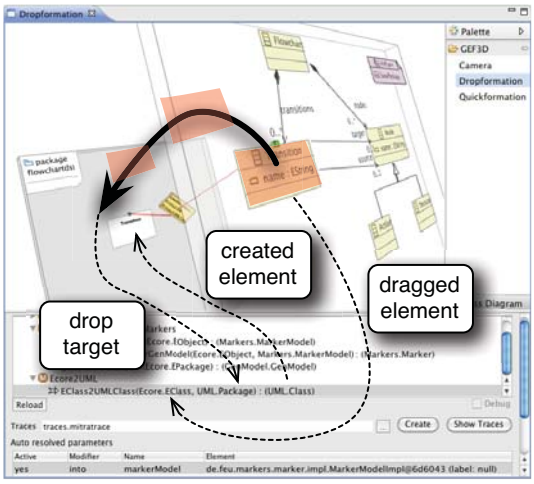


Figure 1: Dropformation

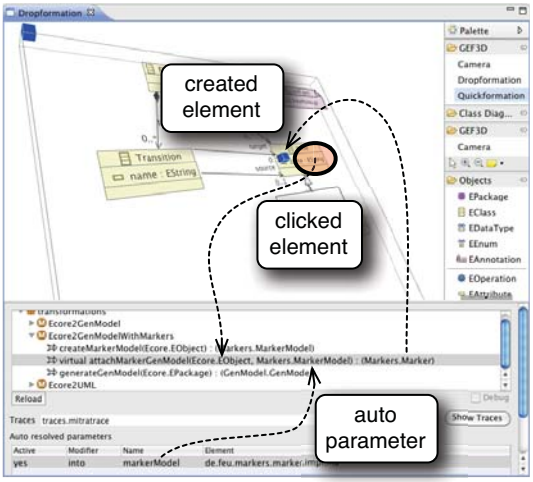


Figure 2: Quickformation

The second concept is called *quickformation*. In this case, no elements are dragged, instead only one argument is provided by the user by simple clicking on the element in the diagram. The following rule creates markers for arbitrary Ecore elements:

```
manual traced virtual attachMarkerGenModel
    (from Ecore.EObject element, into Markers.MarkerModel markerModel):
    (create Markers.Marker marker)
```

Since only one argument (the **from** parameter) is provided, the other argument has to be specified differently. In the example, the second parameter (the **into** parameter) is defined as an *auto (resolved) parameter*. Whenever a quickformation has to be executed, the engine firstly binds selected elements, and then it tries to bind missing parameters using specified auto parameters.[5)]

The diagram presentations of the models is usually very useful for users of the transformation, while transformation developers might prefer working with a representation of the abstract syntax tree, e.g., provided by the sample Ecore Model editor. For that purpose, Mitra provides a so called *Mitra Control* view. Just as in the Dropformation view, rules can be selected and trace models can be loaded and created. Instead of using drop- or quick-formations, rule parameters are presented in a table. The parameters can be set by simply dragging elements from the Ecore Model editor onto the parameter row.

[5)] In the marker example, the marker model could also be resolved automatically using traces and triggers as decribed in subsection 2.3. The auto parameter mechanism is used here mainly for demonstration purposes.

2. Solution of the Ecore2GenModel Case

2.1. Using Annotations

Although Mitra is specialized on semi-automated transformations, it can be used for fully automated transformations as well. The first transformation is a 1:1 copy of the ETL transformation attached to the case study. Just as proposed in the case study, this first transformation copies values defined in annotations to the GenModel elements. Annotations can be set using the properties view of the Ecore Tools editor (embedded into the 3D editor).

The transformation itself looks very much like the Epsilon Transformation Language (ETL) version. Mitra is almost strictly imperative, thus rules must contain code to transform contained elements as the model tree is not traversed automatically. The following loop shows a snippet from the package rule:

```
forEach (var Ecore.EClassifier eClassifier in s.eClassifiers) {
    EClassifiers2GenClassifiers(eClassifier, t);
}
```

While the model tree must be traversed programmatically, reverse lookups are not nessary. Containers are passed as **into**-parameters, which probably makes the code much easier readable for Java experienced programmers. Just as in Java, rule can be overloaded. This feature is used in order to overload the rule transforming classifiers:

```
called virtual EClassifiers2GenClassifiers(from Ecore.EClassifier s,
    into GenModel.GenPackage genPackage):  (return GenModel.GenClassifier t) ...

called EClass2GenClass(from Ecore.EClass s, into GenModel.GenPackage genPackage):
    (create GenModel.GenClass t)
        overloads EClassifiers2GenClassifiers(Ecore.EClassifier s, GenModel.GenPackage
            genPackage): (GenModel.GenClassifier t) ...

called EEnum2GenEnum(from Ecore.EEnum s, into GenModel.GenPackage genPackage):
    (create GenModel.GenEnum t)
        overloads EClassifiers2GenClassifiers(Ecore.EClassifier s, GenModel.GenPackage
            genPackage): (GenModel.GenClassifier t) ...
```

2.2. Reflection

The annotations are copied using the reflection features of Mitra. The following (simplified) snippet shows how features are set using reflection. The conversion is done by calling a native method `parse()` (which can be applied on all types).[6)]

[6)] Mitra distinguished between `target.feature` and `target.<<feature>>`. In the first case, `feature` is the name of a field, while in the second case, `<<feature>>` is an expression. As in Object Constraint Language (OCL), Mitra distinguishes between fields of a model type (using the dot notation) and native methods (using the arrow notation).

```
public called setFeature(any target, String feature, any value) {
    if (target.<<feature>>—>isAssignableFrom(value)) { target.<<feature>> = value; }
    else {
        var any parsed = target.<<feature>>—>type()—>parse(value);
        if (parsed!=null) { target.<<feature>> = parsed; }
    }
}
```

2.3. Markers

Annotations can only be used if the source model supports them. Unfortunately, this is not always the case. Thus we suggest an alternative solution, using markers instead of annotations. In order to use markers, firstly a marker model has to be created using a transformation rule as well. Then, new markers can be added to model elements using quickformations as described above (see Figure 2). In order to provide an information to the user about what features can be set, markers are populated with all features available in the target GenModel elements. This is also done with reflection:

```
called populateMarker(use Type type, into Markers.Marker marker) {
    forEach(var String featureName in type—>fieldNames()) {
        if (type—>fieldChangeable(featureName)) {
            var Type fieldType = type—>fieldType(featureName);
            if (fieldType—>canParse()) { setProperty(marker, featureName, null); }
}   }   }
```

In order to resolve a marker associated to a specific element, Mitra uses *triggers*. Triggers are much like database triggers, however they are not activated by model changes but by certain states of the transformation. Mitra natively supports persistable traces, which are usually used for saving the state of the transformation in order to allow multi-sessions when semi-automatically transforming models.[7] These traces are used to trigger rules automatically. The following code snippet shows a rule which is automatically triggered when an Ecore element is transformed to a GenModel element *and* a marker has been set before:

```
auto copyMarkerProperties(from Markers.Marker marker, into any target)
    trigger (
        EcoreToGenModel(any element, any genmodel_container): (any target),
        attachMarkerGenModel(Ecore.EObject element, Markers.MarkerModel markerModel):
            (Markers.Marker marker)
    ) { /* populate genmodel features with marker properties */     }
```

Triggers can be interpreted as a declarative extension to the otherwise imperative language. The parameters of the rule are to be set by the trigger (or an optional **with**-clause). In the example, the parameters `marker` and `target` are bound to values provided by the rules specified in the trigger. An implicit when-clause clause is defined by similar parameter names, which implies equal values (optionally, an explicit **when**-clause can be defined).

[7] In order to create a trace, a rule must be annotated with the modifier **traced**. Traced rules are also cached, that is when a traced rule is called with the same input parameters multiple times, it is only executed once.

2.4. 3D Benefits

So far, using 3D in the graphical view might be nice but it is not really necessary. Instead of creating 3D cubes, markers could have been visualized with blue rectangles in a 2D view as well. As we noted in section 1, Mitra's main focus is on semi-automated M2M-transformation whereby visualization of multiple diagrams is required. Besides 3D, GEF3D also provides special techniques for combining multiple editors. This is needed for combining the Ecore Tools editor and the marker editor. To give an impression of the benefits of 3D, a small transformation example is included for transforming Ecore classes to UML classes (see Figure 1). This transformation does not only show a typical CAT, but also how traces are visualized with GEF3D (see Figure 3).

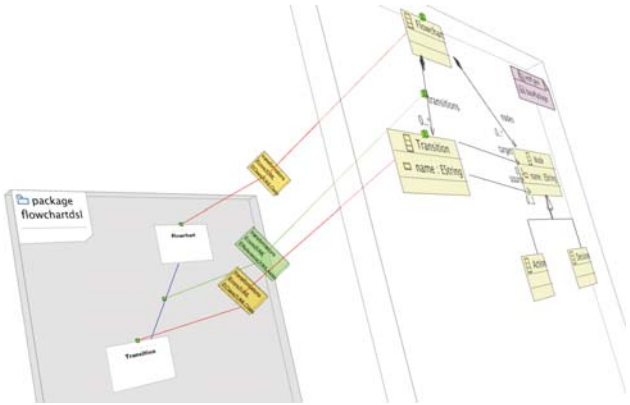


Figure 3: Visualization of traces

3. Discussion

Using the reflection features of Mitra, the case study example could be implemented similarly to the solution proposed by Kolovos et al. Our solution does not use traces for updating the target model automatically, however traces are used by the transformation itself. Although reflection is a powerful mechanism, it disables type checking. E.g., using the non-reflective method of setting a field immediately shows an error if the wrong field name is used. The case study revealed some weaknesses of Metra when handling collections (OCL support is planned but not yet realized) and enumerations.

Providing a good transformation language is only one side of the story. The other side of the story is to provide a good user interface. Since many modeling languages use a graphical notation, transformation tools should (re-) use the concrete notation, i.e. diagrams, as well. GEF3D provides a lot of mechanisms in order to simplify the 3D-fication of existing editors (note that the 2D-diagrams are still fully editable). However, adding transformation support remains a challenge. E.g., the Ecore Tools editor does not automatically reflect model changes while the UML2 Tools editors do. Note that the submitted tool provides only a "raw" drop- and quickformation user interface. For specific applications, many settings could be defined automaticlly, such as predefinded trace models, smart selection of rules (based on the parameter types and other context information), or other domain specific improvments.[8)]

[8)] An example of a domain specific application, a graphical editor for the GMF mapping model using Mitra and GEF3D can be found at `http://dev.eclipse.org/blogs/gef3d/2010/01/20/a-graphical-editor-for-the-gmf-mapping-model/`.

Appendix

A. Listing Ecore2GenModel

This first listing shows the transformation discussed in subsection 2.1. It mainly consits of two parts:

- rules transforming ecore elements to genmodel elements
- helper rules

This is a rather long listing, and usually this would be split up into several modules (Mitra supports include/import of modules).

```
/*
Transformation Tool Contest 2010
Ecore to GenModel (cf 2.1 Using Annotations)
*/
module transformations:Ecore2GenModel {

    /* Declaration of metamodels */
    metamodel ecore:Ecore (nsUri="http://www.eclipse.org/emf/2002/Ecore");
    metamodel ecore:GenModel (nsUri="http://www.eclipse.org/emf/2002/GenModel");

    /* One and only MANUAL rule, which is to be invoked manually using the Quickformation
        mechanism.
        Remarks:
        — the parameter modifier create let Mitra automatically create a new instance; other
            instances are to be created using a Java—like constructor
        — Mitra supports mulitple return parameters, however in this example this is not used
            (but it is helpful if n source elements are transformed to m target elements)
    */
    manual generateGenModel(from Ecore.EPackage s) : (create GenModel.GenModel genModel) {
        var GenModel.GenPackage genPackage = EPackage2GenPackage(s, genModel);
        genModel.genPackages += genPackage;         // add newly created package to container

        // set with defaults
        setAnnotation(s, genModel, "complianceLevel", $GenModel.GenJDKLevel.JDK60);
        setAnnotation(s, genModel, "copyrightFields", false);
        setAnnotation(s, genModel, "modelPluginID", s.name);
        setAnnotation(s, genModel, "modelDirectory", "/" + s.name + "/src");
        setAnnotation(s, genModel, "modelName", s.name—>firstToUpperCase());
        setAnnotation(s, genModel, "importerID", "org.eclipse.emf.importer.ecore");

        // copy others
        copyAnnotations(s, genModel);
    }


```

A. Listing Ecore2GenModel

```
37  called EPackage2GenPackage(from Ecore.EPackage s, into GenModel.GenModel genModel):
38          (create GenModel.GenPackage t) {
39      genModel.genPackages += t;
40      t.ecorePackage = s;
41      setAnnotation(s, t, "disposableProviderFactory", true);
42      setAnnotation(s, t, "prefix", s.name->firstToUpperCase());
43      copyAnnotations(s, t);
44
45      forEach (var Ecore.EClassifier eClassifier in s.eClassifiers) {
46          EClassifiers2GenClassifiers(eClassifier, t);
47      }
48  }
49
50  /* A virtual rule can be overloaded. Since EClassifier is an abstract class, this rule does
51      not get invoked directly.
52  */
53  called virtual EClassifiers2GenClassifiers(from Ecore.EClassifier s,
54          into GenModel.GenPackage genPackage): (return GenModel.GenClassifier t) {
55  }
56
57  /* This rule overloads the afore defined rule. In Mitra, overloading a rule (or
58      implementing abstract rules) is more flexibel as in other OO languages:
59      The rule names (of overloaded/implemented and overloading/implementing rule)
60      may differ, and it is even possible to overload/implement a rule with
61      a rule having a different number of parameters. However, this special feature is
62      not used in the example.
63  */
64  called EClass2GenClass(from Ecore.EClass s, into GenModel.GenPackage genPackage):
65          (create GenModel.GenClass t)
66      overloads EClassifiers2GenClassifiers(Ecore.EClassifier s,
67          GenModel.GenPackage genPackage):    (GenModel.GenClassifier t)
68  {
69      genPackage.genClasses += t;
70      t.ecoreClass = s;
71
72      setAnnotation(s, t, "image", ! s.^abstract);
73      copyAnnotations(s, t);
74
75      forEach (var Ecore.EStructuralFeature eStructuralFeature in s.eStructuralFeatures) {
76          EStructuralFeature2GenFeature(eStructuralFeature, t);
77      }
78      forEach (var Ecore.EOperation eOperation in s.eOperations) {
79          EOperation2GenOperation(eOperation, t);
80      }
81  }
82
83  called virtual EStructuralFeature2GenFeature(from Ecore.EStructuralFeature s,
84          into GenModel.GenClass genClass): (create GenModel.GenFeature t) {
85      genClass.genFeatures += t;
86      t.ecoreFeature = s;
87  }
```

```
88
89      /* This rules overloads the virtual rule above. The overloaded rule can be called from an
90          overloading rule by using the statement 'super'. Just as in Java constructors, the
91          overloaded  rule must be called at the very beginning of the body of the overloading
92          rule. In contrast to    constructors in Java, the overloading rule is only called with
93          super and not automatically.
94
95          '$' refers to the type in expressions. E.g., '$GenModel.GenPropertyKind' refers to the
96          type itself. This way, static fields and methods can be called. It is even possible to
97          store the   type in a variable, as demonstrated later on.
98      */
99      called EAttribute2GenFeature(from Ecore.EAttribute s, into GenModel.GenClass genClass):
100             (create GenModel.GenFeature t)
101         overloads EStructuralFeature2GenFeature(Ecore.EStructuralFeature s,
102             GenModel.GenClass genClass): (GenModel.GenFeature t)
103     {
104         super; // calls overloaded rule
105
106         var GenModel.GenPropertyKind defaultProperty;
107         if (s.changeable) {
108             defaultProperty = $GenModel.GenPropertyKind.Editable; // static access
109         } else {
110             defaultProperty = $GenModel.GenPropertyKind.Readonly;
111         }
112
113         setAnnotation(s, t, "children", false);
114         setAnnotation(s, t, "createChild", false);
115         setAnnotation(s, t, "notify", true);
116         setAnnotation(s, t, "propertySortChoices", false);
117         setAnnotation(s, t, "property", defaultProperty);
118         copyAnnotations(s, t);
119     }
120
121     called EReference2GenFeature(from Ecore.EReference s, into GenModel.GenClass genClass):
122             (create GenModel.GenFeature t)
123         overloads EStructuralFeature2GenFeature(Ecore.EStructuralFeature s,
124             GenModel.GenClass genClass): (GenModel.GenFeature t)
125     {
126         super;
127
128         var GenModel.GenPropertyKind defaultProperty;
129         if (! s.container &&  ! s.containment) {
130             if (s.changeable) {
131                 defaultProperty = $GenModel.GenPropertyKind.Editable;
132             }
133             else {
134                 defaultProperty = $GenModel.GenPropertyKind.Readonly;
135             }
136         }
137         else {
138             defaultProperty = $GenModel.GenPropertyKind.None;
```

A. Listing Ecore2GenModel

```
139         }
140
141         setAnnotation(s, t, "children", s.containment);
142         setAnnotation(s, t, "createChild", t.children && s.changeable);
143         setAnnotation(s, t, "notify", t.children);
144         setAnnotation(s, t, "propertySortChoices",
145             defaultProperty == $GenModel.GenPropertyKind.Editable);
146         setAnnotation(s, t, "property", defaultProperty);
147         copyAnnotations(s, t);
148     }
149
150     called EOperation2GenOperation(from Ecore.EOperation s, into GenModel.GenClass genClass):
151                 (create GenModel.GenOperation t) {
152             genClass.genOperations += t;
153             t.ecoreOperation = s;
154         copyAnnotations(s, t);
155
156         forEach (var Ecore.EParameter eParamter in s.eParameters) {
157             EParameter2GenParameter(eParamter, t);
158         }
159     }
160
161     called EParameter2GenParameter(from Ecore.EParameter s,
162         into GenModel.GenOperation genOperation):   (create GenModel.GenParameter t) {
163         genOperation.genParameters += t;
164         t.ecoreParameter = s;
165         copyAnnotations(s, t);
166     }
167
168     called EEnum2GenEnum(from Ecore.EEnum s, into GenModel.GenPackage genPackage):
169             (create GenModel.GenEnum t)
170         overloads EClassifiers2GenClassifiers(Ecore.EClassifier s,
171             GenModel.GenPackage genPackage):    (GenModel.GenClassifier t)
172     {
173         genPackage.genEnums += t;
174         t.ecoreEnum = s;
175
176         setAnnotation(s, t, "typeSafeEnumCompatible", false);
177         copyAnnotations(s, t);
178
179         forEach (var Ecore.EEnumLiteral eLiteral in s.eLiterals) {
180             EEnumLiteral2GenEnumLiteral(eLiteral, t);
181         }
182     }
183
184     called EEnumLiteral2GenEnumLiteral(from Ecore.EEnumLiteral s,
185             into GenModel.GenEnum genEnum): (create GenModel.GenEnumLiteral t)  {
186         genEnum.genEnumLiterals += t;
187         t.ecoreEnumLiteral = s;
188         copyAnnotations(s, t);
189     }
```

A. Listing Ecore2GenModel

```
190
191     called EDataType2GenDataType(from Ecore.EDataType s,
192             into GenModel.GenPackage genPackage): (create GenModel.GenDataType t)
193             overloads EClassifiers2GenClassifiers(Ecore.EClassifier s,
194             GenModel.GenPackage genPackage):    (GenModel.GenClassifier t)
195     {
196         genPackage.genDataTypes += t;
197         t.ecoreDataType = s;
198         copyAnnotations(s, t);
199     }
200
201     /*-------------------------------------------------------------------------------
202         Helper rules, called by the rules above:
203     */
204
205     called  setAnnotation(from Ecore.EModelElement source, into any target, String label,
206             any default) {
207         var any value = getAnnotation(source, label);
208         if (value==null) value = default;
209         setFeature(target, label, value);
210     }
211
212     /* This rule uses reflection in order to set a feature of a target.
213        — '<<..>>' indicates an expression within a feature access statement.
214        — '—>' indicates a native method call, in contrast to operation calls using
215             the dot notation. This is similar to OCL
216     */
217     public called setFeature(any target, String feature, any value) {
218         if (target.<<feature>>—>isAssignableFrom(value)) {
219             target.<<feature>> = value;
220         } else {
221             if (target.<<feature>>—>isMany()) {
222                 forEach (var String part in value—>split(",")) {
223                     target.<<feature>>+= part;
224                 }
225             } else {
226                 var any parsed = target.<<feature>>—>type()—>parse(value);
227                 if (parsed!=null) { target.<<feature>> = parsed; }
228             }
229         }
230     }
231
232     called copyAnnotations(from Ecore.EModelElement source, into any target) {
233         forEach (var String feature in target—>fieldNames()) {
234             var any value = getAnnotation(source, feature);
235             if (value!=null) {
236                 setFeature(target, feature, value);
237             }
238         }
239     }
240
```

```
241     /* The select statement used in this rule works quite similar to select in
242        OCL or SQL.
243     */
244     called getAnnotation(Ecore.EModelElement element,  String label) : (return any ret) {
245         var Ecore.EAnnotation ann = select first (
246             var Ecore.EAnnotation a in element.eAnnotations where a.source == "emf.gen");
247         if (ann!=null) {
248             var Ecore.EStringToStringMapEntry det = select first (
249                 var Ecore.EStringToStringMapEntry d in ann.details where d.key == label);
250             if (det!=null)
251                 return det.value;
252         }
253         return null;
254     }
255 }
```

B. Listing Ecore2GenModelWithMarkers

This second listing shows the transformation discussed in subsection 2.3. It mainly consits of five parts:

- rules creating the markers
- helper rules for creating markers
- triggered rules to populate features of GenModel element with values of marker properties
- rules transforming ecore elements to genmodel elements
- helper rules for transformation

```
1  /*
2  Transformation Tool Contest 2010
3  Ecore to GenModel with markers (cf. 2.3 Markers)
4  */
5  module transformations:Ecore2GenModelWithMarkers {
6      metamodel ecore:Ecore (nsUri="http://www.eclipse.org/emf/2002/Ecore");
7      metamodel ecore:GenModel (nsUri="http://www.eclipse.org/emf/2002/GenModel");
8      metamodel ecore:Markers (nsUri="http://feu.de/marker");
9
10     /* This rule simply creates the marker model. Note that in the submission, this
11         rule is called manually and the created marker model is specified as an auto
12         parameter. Since traced rules are cached (that is they are only executed the
13         first time when called multiple times with the same input parameters),
14         it would have been possible to call this rule from the following rules in order
15         to automaticlly retrieve the marker model.
16     */
17     manual traced createMarkerModel(use Ecore.EObject element):
18             (create Markers.MarkerModel markerModel) {
19         markerModel.element = element;
20     }
```

B. Listing Ecore2GenModelWithMarkers

```
21      /*
22          This virual rule is overloaded by rules for specific types.
23          From the user's point of view, only manual rules are visible, that is only this
24          virtual rule can be selected in the user interface. Also, only this base rule is used
25          later on in the trigger.
26      */
27      manual traced virtual attachMarkerGenModel(from Ecore.EObject element,
28              into Markers.MarkerModel markerModel): (create Markers.Marker marker)
29      {
30          marker.element = element;
31          markerModel.markers += marker;
32      }
33
34      /*
35          Overloads afore declared virtual rule, uses 'super' to call the overloading rule.
36      */
37      called traced attachMarkerGenModel(use Ecore.EPackage package,
38              into Markers.MarkerModel markerModel): (create Markers.Marker marker)
39          overloads attachMarkerGenModel(Ecore.EObject package,
40              Markers.MarkerModel markerModel): (Markers.Marker marker)
41      {
42          super; // call overloading rule
43
44          // Note: $GenModel.GenPackage returns the type itself
45          populateMarker($GenModel.GenPackage, marker); // populate marker properties
46          populateMarker($GenModel.GenModel, marker); // also with model features
47
48          // set default values
49          setProperty(marker, "complianceLevel", $GenModel.GenJDKLevel.JDK60);
50          setProperty(marker, "copyrightFields", false);
51          setProperty(marker, "modelPluginID", package.name);
52          setProperty(marker, "modelDirectory", "/" + package.name + "/src");
53          setProperty(marker, "modelName", package.name->firstToUpperCase());
54          setProperty(marker, "importerID", "org.eclipse.emf.importer.ecore");
55      }
56
57      called traced attachMarkerGenModel(use Ecore.EClass s,
58              into Markers.MarkerModel markerModel): (create Markers.Marker marker)
59          overloads attachMarkerGenModel(Ecore.EObject, Markers.MarkerModel):
60              (Markers.Marker)
61      {
62          super;
63          populateMarker($GenModel.GenClass, marker);
64          setProperty(marker, "image", ! s.^abstract);
65      }
66
67
68
69
70
71
```

B. Listing Ecore2GenModelWithMarkers

```
72      called traced attachMarkerGenModel(use Ecore.EAttribute s,
73              into Markers.MarkerModel markerModel): (create Markers.Marker marker)
74          overloads attachMarkerGenModel(Ecore.EObject, Markers.MarkerModel):
75              (Markers.Marker)
76      {
77          super;
78          populateMarker($GenModel.GenFeature, marker);
79
80          var GenModel.GenPropertyKind defaultProperty;
81          if (s.changeable) {
82              defaultProperty = $GenModel.GenPropertyKind.Editable;
83          } else {
84              defaultProperty = $GenModel.GenPropertyKind.Readonly;
85          }
86          setProperty(marker, "children", false);
87          setProperty(marker, "createChild", false);
88          setProperty(marker, "notify", true);
89          setProperty(marker, "propertySortChoices", false);
90          setProperty(marker, "property", defaultProperty);
91      }
92
93      called traced attachMarkerGenModel(use Ecore.EReference s,
94              into Markers.MarkerModel markerModel): (create Markers.Marker marker)
95          overloads attachMarkerGenModel(Ecore.EObject, Markers.MarkerModel):
96              (Markers.Marker)
97      {
98          super;
99          populateMarker($GenModel.GenFeature, marker);
100
101         var GenModel.GenPropertyKind defaultProperty;
102         if (! s.container &&  ! s.containment) {
103             if (s.changeable) {
104                 defaultProperty = $GenModel.GenPropertyKind.Editable;
105             }
106             else {
107                 defaultProperty = $GenModel.GenPropertyKind.Readonly;
108             }
109         }
110         else {
111             defaultProperty = $GenModel.GenPropertyKind.None;
112         }
113
114         setProperty(marker, "children", s.containment);
115         setProperty(marker, "createChild", s.containment && s.changeable);
116         setProperty(marker, "notify", s.containment);
117         setProperty(marker, "propertySortChoices",
118             defaultProperty == $GenModel.GenPropertyKind.Editable);
119         setProperty(marker, "property", defaultProperty);
120     }
121
122
```

```
123     called traced attachMarkerGenModel(use Ecore.EOperation s,
124             into Markers.MarkerModel markerModel): (create Markers.Marker marker)
125         overloads attachMarkerGenModel(Ecore.EObject, Markers.MarkerModel):
126             (Markers.Marker)
127     {
128         super;
129         populateMarker($GenModel.GenOperation, marker);
130     }
131
132     called traced attachMarkerGenModel(use Ecore.EParameter s,
133         into Markers.MarkerModel markerModel): (create Markers.Marker marker)
134         overloads attachMarkerGenModel(Ecore.EObject, Markers.MarkerModel): (Markers.Marker)
135     {
136         super;
137         populateMarker($GenModel.GenParameter, marker);
138     }
139
140     called traced attachMarkerGenModel(use Ecore.EEnum s,
141             into Markers.MarkerModel markerModel): (create Markers.Marker marker)
142         overloads attachMarkerGenModel(Ecore.EObject, Markers.MarkerModel):
143             (Markers.Marker)
144     {
145         super;
146         populateMarker($GenModel.GenEnum, marker);
147         setProperty(marker, "typeSafeEnumCompatible", false);
148     }
149
150     called traced attachMarkerGenModel(use Ecore.EDataType s,
151             into Markers.MarkerModel markerModel): (create Markers.Marker marker)
152         overloads attachMarkerGenModel(Ecore.EObject, Markers.MarkerModel):
153             (Markers.Marker)
154     {
155         super;
156         populateMarker($GenModel.GenDataType, marker);
157     }
158
159
160     /*-------------------------------------------------------------------------------
161         Helper rules for attaching markers:
162     */
163
164     called populateMarker(use Type type, into Markers.Marker marker) {
165         forEach(var String featureName in type->fieldNames()) {
166             if (type->fieldChangeable(featureName)) {
167                 var Type fieldType = type->fieldType(featureName);
168                 if (fieldType->canParse()) {
169                     setProperty(marker, featureName, null);
170                 }
171             }
172         }
173     }
```

B. Listing Ecore2GenModelWithMarkers

```
174  /*------------------------------------------------------------------------------
175      Automatically triggered rules, these rules populate the GenModel element's
176      features with the property values of the marker.
177  */
178
179  /* This rule is triggered by the rules defined in the trigger section. Since this rule is
180      called automatically, all parameters must be bind by in the trigger. That is,
181     - marker is bound using the return value of attachMarkerGenModel(..)
182     - target is bound using the return value of EcoreToGenModel(..)
183      Note that both trigger rules have a parameter 'element'. That is, the values of
184      this parameter must be equal!
185  */
186  auto copyMarkerProperties(from Markers.Marker marker, into any target)
187  trigger (
188      EcoreToGenModel(any element, any genmodel_container): (any target),
189      attachMarkerGenModel(Ecore.EObject element, Markers.MarkerModel markerModel):
190          (Markers.Marker marker)
191  ) {
192      out("copy marker properties into " + target); // only for debugging purposes
193      forEach (var String feature in target->fieldNames()) {
194          var Markers.StringProperty property =   select first (
195              var Markers.StringProperty p in marker.properties where p.name==feature);
196
197          out("property for feature " + feature + " is " + property);
198
199          if (property!=null && property.stringValue!=null) {
200              out("set feature " + feature + " from marker " + property.stringValue);
201
202              setFeature(target, feature, property.stringValue);
203          }
204      }
205  }
206
207  /* A second automatically called rule. Note that the second rule in the trigger equals the
208      second rule in the trigger defined in the rule above. A single traced rule can used in
209      several triggers. Also note that the order of the execution of the triggering rules (as
210      the order of the definition in the trigger) does not matter.
211  */
212  auto copyMarkerPropertiesModel(from Markers.Marker marker, into GenModel.GenModel genModel)
213  trigger (
214      EPackage2GenPackage(Ecore.EPackage epackage, GenModel.GenModel genModel):
215          (GenModel.GenPackage t),
216      attachMarkerGenModel(Ecore.EPackage epackage, Markers.MarkerModel markerModel):
217          (Markers.Marker marker)
218  ) {
219      out("extra: Model: copy marker properties into " + genModel);
220      forEach (var String feature in genModel->fieldNames()) {
221          var Markers.StringProperty property =
222              select first (var Markers.StringProperty p in marker.properties where p.name==
                     feature);
223
```

B. Listing Ecore2GenModelWithMarkers

```
            out("property for feature " + feature + " is " + property);

            if (property!=null && property.stringValue!=null) {
                out("set feature " + feature + " from marker " + property.stringValue);

                setFeature(genModel, feature, property.stringValue);
            }
        }
    }

    called setProperty(into Markers.Marker marker, String name, any value) {
        var Markers.StringProperty property =   select first (
            var Markers.StringProperty p in marker.properties where p.name==name);
        if (property==null) {
            property = new Markers.StringProperty();
            property.name = name;
            marker.properties += property;
        }
        if (value!=null)
            property.stringValue = value->toString();

    }

    public called setFeature(any target, String feature, any value) {
        if (target.<<feature>>->isAssignableFrom(value)) {
            target.<<feature>> = value;
        } else {
            if (target.<<feature>>->isMany()) {
                forEach (var String part in value->split(",")) {
                    target.<<feature>>+= part;
                }
            } else {
                var any parsed = target.<<feature>>->type()->parse(value);
                if (parsed!=null) {
                    target.<<feature>> = parsed;
                }
            }
        }
    }


    /*-------------------------------------------------------------------------------
        Transformation rules, quite similar to first transformation, except that
        markers are used instead of annotations.
    */

    /* Abstract rule, is implemented by concrete rules later on. This abstract rule is defined
        in order to simplify the definition of the triggers.
    */
    abstract traced EcoreToGenModel(from any ecore_element, into any genmodel_container):
        (return any genmodel_element);
```

B. Listing Ecore2GenModelWithMarkers

```
275  /* Just as in the first example the rule to actually create the GenModel */
276  manual generateGenModel(from Ecore.EPackage s) : (create GenModel.GenModel genModel) {
277      var GenModel.GenPackage genPackage = EPackage2GenPackage(s, genModel);
278      genModel.genPackages += genPackage;
279
280      // set with defaults
281      genModel.complianceLevel = $GenModel.GenJDKLevel.JDK60;
282      genModel.copyrightFields = false;
283      genModel.modelPluginID = s.name;
284      genModel.modelDirectory = "/" + s.name + "/src";
285      genModel.modelName = s.name->firstToUpperCase();
286      genModel.importerID = "org.eclipse.emf.importer.ecore";
287  }
288
289  /* Called from afore defined rule. The rules implements the abstract rule in order to
290      trigger the auto rules. This pattern is true for all following rules.
291  */
292  called traced EPackage2GenPackage(from Ecore.EPackage s, into GenModel.GenModel genModel):
293          (create GenModel.GenPackage t)
294      implements  EcoreToGenModel(any, any ): (any)
295  {
296      genModel.genPackages += t;
297      t.ecorePackage = s;
298      t.disposableProviderFactory = true;
299      t.prefix = s.name->firstToUpperCase();
300      forEach (var Ecore.EClassifier eClassifier in s.eClassifiers) {
301          EClassifiers2GenClassifiers(eClassifier, t);
302      }
303  }
304
305  /* Just as in the previous listing, this virtual rule cannot be called as EClassifier is
306      abstract.
307  */
308  called traced virtual EClassifiers2GenClassifiers(from Ecore.EClassifier s,
309          into GenModel.GenPackage genPackage): (return GenModel.GenClassifier t)
310      implements  EcoreToGenModel(any, any ): (any)
311  {
312      // must not be called
313  }
314
315  called EClass2GenClass(from Ecore.EClass s, into GenModel.GenPackage genPackage):
316          (create GenModel.GenClass t)
317      overloads EClassifiers2GenClassifiers(Ecore.EClassifier s,
318          GenModel.GenPackage genPackage):    (GenModel.GenClassifier t)
319  {
320      genPackage.genClasses += t;
321      t.ecoreClass = s;
322      t.image = ! s.^abstract;
323
324      forEach (var Ecore.EStructuralFeature eStructuralFeature in s.eStructuralFeatures) {
325          EStructuralFeature2GenFeature(eStructuralFeature, t);
```

```
326         }
327         forEach (var Ecore.EOperation eOperation in s.eOperations) {
328             EOperation2GenOperation(eOperation, t);
329         }
330     }
331
332     called traced virtual EStructuralFeature2GenFeature(from Ecore.EStructuralFeature s,
333             into GenModel.GenClass genClass): (create GenModel.GenFeature t)
334         implements  EcoreToGenModel(any, any ): (any)
335     {
336         genClass.genFeatures += t;
337         t.ecoreFeature = s;
338     }
339
340     called EAttribute2GenFeature(from Ecore.EAttribute s, into GenModel.GenClass genClass):
341             (create GenModel.GenFeature t)
342     overloads EStructuralFeature2GenFeature(Ecore.EStructuralFeature s,
343             GenModel.GenClass genClass): (GenModel.GenFeature t)
344     {
345         super;
346
347         var GenModel.GenPropertyKind defaultProperty;
348         if (s.changeable) {
349             defaultProperty = $GenModel.GenPropertyKind.Editable;
350         } else {
351             defaultProperty = $GenModel.GenPropertyKind.Readonly;
352         }
353
354         t.children = false;
355         t.createChild = false;
356         t.notify = true;
357         t.propertySortChoices = false;
358         t.property = defaultProperty;
359     }
360
361     called EReference2GenFeature(from Ecore.EReference s, into GenModel.GenClass genClass):
362             (create GenModel.GenFeature t)
363         overloads EStructuralFeature2GenFeature(Ecore.EStructuralFeature s,
364             GenModel.GenClass genClass): (GenModel.GenFeature t)
365     {
366         super;
367
368         var GenModel.GenPropertyKind defaultProperty;
369         if (! s.container &&  ! s.containment) {
370             if (s.changeable) {
371                 defaultProperty = $GenModel.GenPropertyKind.Editable;
372             }
373             else {
374                 defaultProperty = $GenModel.GenPropertyKind.Readonly;
375             }
376         }
```

B. Listing Ecore2GenModelWithMarkers

```
377         else {
378             defaultProperty = $GenModel.GenPropertyKind.None;
379         }
380 
381         t.children = s.containment;
382         t.createChild = t.children && s.changeable;
383         t.notify = t.children;
384         t.propertySortChoices = defaultProperty == $GenModel.GenPropertyKind.Editable;
385         t.property = defaultProperty;
386 
387     }
388 
389     called traced EOperation2GenOperation(from Ecore.EOperation s,
390             into GenModel.GenClass genClass): (create GenModel.GenOperation t)
391             implements  EcoreToGenModel(any, any ): (any)
392     {
393             genClass.genOperations += t;
394             t.ecoreOperation = s;
395         forEach (var Ecore.EParameter eParamter in s.eParameters) {
396             EParameter2GenParameter(eParamter, t);
397         }
398     }
399 
400     called traced EParameter2GenParameter(from Ecore.EParameter s,
401             into GenModel.GenOperation genOperation):   (create GenModel.GenParameter t)
402             implements  EcoreToGenModel(any, any ): (any)
403     {
404         genOperation.genParameters += t;
405         t.ecoreParameter = s;
406     }
407 
408     called traced EEnum2GenEnum(from Ecore.EEnum s, into GenModel.GenPackage genPackage):
409             (create GenModel.GenEnum t)
410         overloads EClassifiers2GenClassifiers(Ecore.EClassifier s,
411                 GenModel.GenPackage genPackage):    (GenModel.GenClassifier t)
412     {
413         genPackage.genEnums += t;
414         t.ecoreEnum = s;
415         t.typeSafeEnumCompatible = false;
416         forEach (var Ecore.EEnumLiteral eLiteral in s.eLiterals) {
417             EEnumLiteral2GenEnumLiteral(eLiteral, t);
418         }
419     }
420 
421     called traced EEnumLiteral2GenEnumLiteral(from Ecore.EEnumLiteral s,
422             into GenModel.GenEnum genEnum): (create GenModel.GenEnumLiteral t)
423         implements  EcoreToGenModel(any, any ): (any)
424     {
425         genEnum.genEnumLiterals += t;
426         t.ecoreEnumLiteral = s;
427     }
```

```
428     called EDataType2GenDataType(from Ecore.EDataType s,
429             into GenModel.GenPackage genPackage): (create GenModel.GenDataType t)
430         overloads EClassifiers2GenClassifiers(Ecore.EClassifier s,
431             GenModel.GenPackage genPackage):    (GenModel.GenClassifier t)
432     {
433         genPackage.genDataTypes += t;
434         t.ecoreDataType = s;
435     }
436
437     /* Definition of a native (=Java) method.
438     */
439     called out(String message) native(class="mitra.Log");
440 }
```

C. Listing Ecore2GenModelWithMarkers

This second listing shows the transformation discussed in subsection 2.4. This kind of transformation is used for *dropformations*. In this very simple example, only one manually called rule is defined. However, the strength of Mitra is to enable the definition of different manual rules, in order to allow similar elements to be transformed differently without the need for markers or annotations. Using abstract rules, different rules can then be unified in order to trigger the very same auto rules. As in this example, the auto rules can be used to transform relations between node elements automatically.

```
1  /*
2  Transformation Tool Contest 2010
3  Ecore2UML (cf. 2.4 3D Benefits)
4  */
5  module transformations:Ecore2UML {
6
7      metamodel ecore:Ecore (nsUri="http://www.eclipse.org/emf/2002/Ecore");
8      metamodel uml2:UML ();
9
10     /* The manual rule used in a dropformation. It also triggeres the following rule.
11     */
12     manual traced EClass2UMLClass(from Ecore.EClass s, into UML.Package p) : (create UML.Class
           t) {
13         t.name = s.name;
14         p.packagedElement += t;
15     }
16
17
18
19
20
21
22
23
24
```

C. Listing Ecore2GenModelWithMarkers

```
    /* Triggered by afore defined rule. The with-block defines the first parameter, the second
        parameter is bound to the arguments of the trigger rules.
    */
    auto traced EReference2UMLAssoc(from Ecore.EReference eRef, into UML.Package p):
            (create UML.Association assoc)
    trigger (
        EClass2UMLClass(Ecore.EClass eClassFrom, UML.Package p) : (UML.Class fromClass),
        EClass2UMLClass(Ecore.EClass eClassTo, UML.Package p) : (UML.Class toClass)
    ) with {
        eRef = select first (var Ecore.EReference e in eClassFrom.eReferences where
            e.eType == eClassTo);
    }
    when (eRef!=null ) // explicit when clause
    {
        assoc.name = eRef.name;
        setAssocEnds(assoc, fromClass, toClass);
        p.packagedElement += assoc;
    }

    /* Helper rule
    */
    called setAssocEnds(UML.Association assoc, UML.Class srcClass, UML.Class dstClass) {
        var UML.Property src = new UML.Property();
        src.name = "src";
        src.type = srcClass;
        assoc.ownedEnd += src;

        var UML.Property dst = new UML.Property();
        dst.name = "dst";
        dst.type = dstClass;
        assoc.ownedEnd += dst;
    }
}
```

Ecore to Genmodel case study solution using the Viatra2 framework

Ábel Hegedüs, Zoltán Ujhelyi, Gábor Bergmann, and Ákos Horváth

Budapest University of Technology and Economics, Hungary
{hegedusa,ujhelyiz,bergmann,ahorvath}@mit.bme.hu

Abstract. The paper presents a solution of the Ecore to GenModel case study of the Transformation Tool Contest 2010, using the model transformation tool VIATRA2.

This work was partially supported by the EC FP6 DIANA (AERO1-030985) and ICT FP7 SecureChange (ICT-FET-231101) European Projects.

1 Introduction

Automated model transformations play an important role in modern model-driven system engineering in order to query, derive and manipulate large, industrial models. Since such transformations are frequently integrated to design environments, they need to provide short reaction time to support software engineers.

The objective of the VIATRA2 (VIsual Automated model TRAnsformations [1]) framework is to support the entire life-cycle, i.e. the specification, design, execution, validation and maintenance of model transformations.

Model representation. VIATRA2 uses the VPM metamodeling approach [2] for describing modeling languages and models. The main reason for selecting VPM instead of a MOF-based metamodeling approach is that VPM supports arbitrary metalevels in the model space. As a direct consequence, models taken from conceptually different domains (and/or technological spaces) can be easily integrated into the VPM model space. The flexibility of VPM is demonstrated by a large number of already existing model importers accepting the models of different BPM formalisms, UML models of various tools, XSD descriptions, and EMF models.

Graph transformation (GT) [3] based tools have been frequently used for specifying and executing complex model transformations. In GT tools, *graph patterns* capture structural conditions and type constraints in a compact visual way. At execution time, these conditions need to be evaluated by *graph pattern matching*, which aims to retrieve one or all matches of a given pattern to execute a transformation rule.

Transformation description. Specification of model transformations in VIATRA2 combines the visual, declarative rule and pattern based paradigm of graph transformation (GT) [3] and the very general, high-level formal paradigm of abstract state machines (ASM) [4] into a single framework for capturing transformations within and between modeling languages.

Transformation Execution. Transformations are executed within the framework by using the VIATRA2 interpreter. For pattern matching both (i) *local search based pattern matching* (LS) and (ii) *incremental pattern matching* (INC) are available. This

feature provides the transformation designer additional opportunities to fine tune the transformation either for faster execution (INC) or lower memory consumption (LS) [5].

The rest of the paper is structured as follows. Sec. 2 introduces the Case Study problem which is solved in this paper. Sec. 3 gives an architectural overview of the transformation, while Sec. 4 highlights the interesting parts of our implementation and finally Sec. 5 concludes the paper.

2 Case study

In the Eclipse Modeling Framework (EMF) [6] the Ecore metamodeling language is used for defining arbitrary metamodels. Conforming instance models can be handled by reflection or code generation, the latter is performed by a toolkit provided by EMF. The toolkit first transforms the Ecore metamodel into a GenModel model that stores implementation-specific information and also refers back to the original Ecore metamodel. Then the functional Java classes are generated using a JET-based model-to-text transformation that consumes the GenModel.

As a challenge, [7] proposes the reimplementation of the Ecore to GenModel transformation within a model transformation framework. To overcome the reconciliation problem in the existing transformation (i.e. major changes in the Ecore model can lead to the loss of customised attributes in the existing GenModel) the case study specifies GenModel options as annotations in the source Ecore metamodel. These annotations are populated in the generated GenModel by the transformation.

Furthermore, the case study proposes the application of reflection for handling annotations using a generic function instead of explicit one-by-one mapping for each annotation/attribute pair.

The proposed transformation demonstrates two useful features of transformation languages: (1) ability to establish cross-model references and (2) support for reflection.

3 Solution Architecture

We implemented our solution for the case study using the VIATRA2 model transformation framework. Fig. 1 shows the complete architecture with both preexisting and newly created components. The *Transformation Controller* is an extension to the Eclipse framework that provides an easy-to-use graphical interface for executing the underlying transformation (i.e. it appears as a command in the pop-up menu of Ecore EMF files). From the user perspective, the controller is invoked on an *input Ecore* file and the result is an *output GenModel* file.

Note that the transformation is performed on models *inside the VPM modelspace* of VIATRA2 rather than on in-memory EMF models. Although VIATRA2 does not manipulate EMF models directly, it includes a generic support for handling EMF metamodels and instance models.

In order to understand the transformation we briefly outline the metamodeling approach of our solution. The *Ecore metametamodel* is the base of this support, which was defined in accordance with the actual EMF metamodel of Ecore.

Both the *Ecore and GenModel metamodels* are defined as instances of this metametamodel, and are imported into VIATRA2 with the generic Ecore metamodel importer. Then the input file is used to *import the Ecore file into* VIATRA2 and create the *Ecore model* which is the instance of the Ecore metamodel.

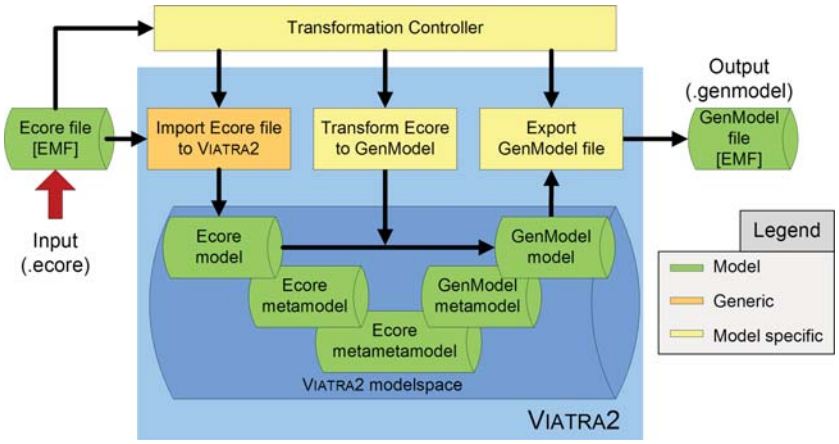


Fig. 1. Solution Architecture

By executing our implemented transformation, we can *transform the Ecore model to a GenModel model* which is an instance of the GenModel metamodel. This *GenModel model* is then *exported* to create the output GenModel file.

Note that currently we have limited generic support for exporting cross-resource references, thus the exporter plugin provided for this case study is GenModel-specific, but we plan to resolve this over time by improving the generic Ecore instance exporter.

4 Transforming Ecore models to GenModels (E2GM)

The *E2GM* transformation generates the GenModel model from the Ecore model in the VIATRA2 framework and is implemented in the VIATRA2 Textual Command Language (VTCL) [8]. E2GM can be separated into two parts, (1) the construction of the GenModel model based on the Ecore model, (2) parsing annotations and creating the corresponding attributes reflectively.

The complete transformation is only 700 lines of VTCL code including whitespaces and comments (see Appendix B). It includes 26 simple type-checking graph patterns (i.e. entity X is type Y, see lines 73-103) and 20 complex patterns (e.g. Ecore annotation that does not have a corresponding GenModel attribute). The type-checking patterns and 10 complex patterns are handled by INC, the other complex patterns by LS. Finally, the actual manipulation is executed by 8 declarative rules (e.g. create GenModel entity for given Ecore entity).

Ecore model traversal is done by navigating through the tree structure of the model and creating GenModel elements with the correct type using a set of mapping key-value pairs (see lines 11-23). The key of the map is the Ecore type (e.g. EClass) and the value is the corresponding GenModel type (GenClass). Although the Ecore metamodel includes a complex type hierarchy (using generalisation), the mapping takes advantage of the fact that for any type in the Ecore model, at most one of its supertypes (or itself) is mapped to a GenModel type (i.e. there is no ambiguity). Therefore, we can define a generic graph pattern that returns the mappable type of any given Ecore element, and this type is used as a key to retrieve the appropriate GenModel type that is instantiated (see lines 172-177).

GenModel and cross-model references. The type of the references between generated GenModel elements are handled similarly to the generic pattern for mapping. Instead of an explicit declaration for every reference type, the Ecore and GenModel metamodels are used (as instances of the Ecore metametamodel) to find the appropriate reference type definition in the GenModel metamodel. Fig. 2 illustrates a graph pattern to find reference type *GenRefType* for containing *GenChild* in *GenParent* (see also lines 208-245).

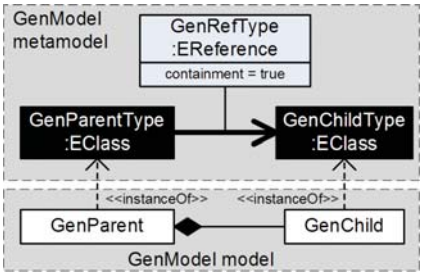


Fig. 2. Pattern for GenModel references

Furthermore, the created GenModel elements have references to the elements in the Ecore model to store the connection between the two models. For example, an *EClass* is transformed into a *GenClass* element in the GenModel with an *ecoreClass* reference between them (see lines 388-399). The transformation creates these references using a generic approach as well (i.e. by retrieving the reference type from the GenModel metamodel).

External Ecore models. EMF provides a capability to create complex interconnected models from more than one Ecore model. Ecore models that are referenced from a source Ecore model are called *external* as their definition is not inside the actual model. Such models are transformed to GenModel models in two different ways: (1) if they already have their own GenModels, then these models are references with the *usedGenPackages* from the generated GenModel; (2) otherwise the generated GenModel will include the packages corresponding to the referenced Ecore models as well. In our E2GM transformation we only support the second case, where new *GenPackages* are created for external Ecore models (see lines 202-206).

Parsing annotations. In our solution we used a reflective approach for parsing annotations in the Ecore model, by retrieving the attributes of GenModel entities from the metamodels using the key of Ecore annotations. Apart from implementing a similar technique proposed in [7] we extended that solution using generic graph patterns to decide the appropriate type for the attributes defined by the annotations. These patterns find the attribute type using the GenModel metamodel, and declarative rules create the attributes themselves (see lines 574-650).

We also implemented type checking for the attributes to ensure that boolean and enumeration values are correct to avoid the generation of a syntactically incorrect GenModel (e.g. boolean with a value other than "true" or "false", see lines 694-724). Furthermore, to provide a GenModel that is usable for code generation without further editing, several default attributes are set even if no annotation is defined for them in the Ecore model (e.g see line 130).

Performance. We used several Ecore models with varying size and complexity to test the performance of our implementation. We tested stand-alone metamodels such as Ecore, XSD, OCL and GTASM (the metamodel for the transformation language of VIATRA2), and metamodels with external Ecore models (e.g. BPEL, UML, WSDL).

As a comparison, we measured[1] the performance of the *built-in EMF generator* on the same metamodels. This generator is a headless Eclipse application which can be invoked from a command prompt. We found that our implementation is *faster on smaller models*, even though we only measured the time the built-in EMF generator required to perform the actual transformation, not the whole time from start up. However, our solution was *slower on larger models* by one order of magnitude (see Fig. 3). The main factors for these results are: (1) presumably, the built-in generator includes an initialisation part with constant time independent of the size of the models; (2) the VIATRA2 framework including a generic pattern matching engine supports a wider range of transformation applications resulting in slower overall execution, compared to the explicit template-based built-in EMF generator.

Stand-alone models	Built-in (ms)	Solution (ms)
Ecore	5875	3047
OCL	6094	3234
XSD	6047	4937
GTASM	5922	16235

With external models	
	Solution (ms)
WSDL	3563
BPEL	6453
UML	36438

Fig. 3. Performance results

It is important to note, that this built-in, headless generator can not handle external Ecore models, therefore we could only test with stand-alone metamodels. Furthermore, we also measured the performance of the transformation in our solution, without the import-export. We found that the execution time is directly proportional to model size, therefore it scales well.

5 Conclusion

In the current paper we have presented our VIATRA2 based implementation for the Ecore to Genmodel case study [7].

Relying on the high-level metamodeling features of VIATRA2, we have presented relatively simple solutions to all optional parts and more. Our implementation is able to handle cross-model references both between several Ecore models and between the generated GenModel and the original Ecore models. We used reflection when dealing with annotations in Ecore models.

The high points of our transformation are the generic rules for mapping Ecore elements to GenModel, which are easily customisable for changes in GenModel metamodel. We exploited the incremental matching feature of VIATRA2 and employed type checking of boolean and enumeration values in annotations. The implementation is able to handle nested packages and external models as well.

On the other hand, import-export of models is required and referenced GenModels (for external Ecore models) are not handled at the moment.

References

1. VIATRA2 Framework: An Eclipse GMT Subproject: (`http://www.eclipse.org/gmt/`)

[1] All measurements were carried out on a computer with Intel Centrino Duo 1.66 GHz processor, 3 GB DDR2 memory, Windows XP, Eclipse 3.5.2 and EMF 2.5.

2. Varró, D., Pataricza, A.: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling **2**(3) (2003) 187–210
3. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific (1999)
4. Börger, E., Stärk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer-Verlag (2003)
5. Bergmann, G., Horváth, A., Ráth, I., Varró, D.: Experimental assessment of combining pattern matching strategies with VIATRA2. Journal of Software Tools in Technology Transfer (2009) Accepted.
6. The Eclipse Modeling Framework project: (http://www.eclipse.org/emf/)
7. Kolovos, D.S., Rose, L.M., Paige, R.F., de Lara, J.: Ecore to GenModel Case Study for TTC2010 (2010)
8. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006), Dijon, France, ACM Press (2006) 1280–1287

A Solution demo and implementation

The deployable implementation and source code is available as an Eclipse online update site (`http://mit.bme.hu/~ujhelyiz/viatra/ttc10-site/`) and as an archive (`http://mit.bme.hu/~ujhelyiz/viatra/ttc10.zip`)

The SHARE image for demonstration purposes is available at `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC10_TTC10%3A%3AXP-Ec2Gm_Viatra_i.vdi`

B Appendix - Ecore to GenModel transformation

```
// metamodel imports
import nemf.packages.ecore;
import nemf.packages.genmodel;
import datatypes;

// Ecore-to-GenModel transformation
@incremental
machine ecore2genmodel {

 // mapping rules for corresponding Ecore and GenModel types
 asmfunction mapping/1 {
  (nemf.packages.ecore.EPackage) = nemf.packages.genmodel.GenPackage;
  (nemf.packages.ecore.ETypeParameter) =
   nemf.packages.genmodel.GenTypeParameter;
  (nemf.packages.ecore.EEnumLiteral) = nemf.packages.genmodel.GenEnumLiteral;
  (nemf.packages.ecore.EDataType) = nemf.packages.genmodel.GenDataType;
  (nemf.packages.ecore.EClass) = nemf.packages.genmodel.GenClass;
  (nemf.packages.ecore.EStructuralFeature) =
   nemf.packages.genmodel.GenFeature;
  (nemf.packages.ecore.EOperation) = nemf.packages.genmodel.GenOperation;
  (nemf.packages.ecore.EParameter) = nemf.packages.genmodel.GenParameter;
  (nemf.packages.ecore.EEnum) = nemf.packages.genmodel.GenEnum;
 }

 // temporal values and entity references
```

7

```
asmfunction temp/1;
// reference to ecore input model
asmfunction ecore/0;
// reference to output genmodel
asmfunction genmodel/0;

// entry point of transformation
// EcoreModel: fully qualified name of the input model
// GenModel: local name of GenModel
// PluginID: plugin identifier of GenModel
rule main(in EcoreModel, in GenModelName, in PluginID) = seq{
 println("[INFO] >>>Ecore2Genmodel Transformation started on " + EcoreModel);
 // find Ecore model in modelspace
 if(ref(EcoreModel) != undef && find EPackage(ref(EcoreModel))) seq{
  update ecore() = ref(EcoreModel);
 } else seq{
  println("[ERROR] EcoreModel not found!");
  fail;
 }
 update temp("pluginID") = PluginID;

 let GenModelRef = "nemf.resources."+GenModelName+"_genmodel" in
 // if GenModel already exists
  if(ref(GenModelRef) != undef &&
   find GenModel(ref(GenModelRef))) seq{
  println("[Warning] Existing GenModel");
  // delete previous GenModel
  delete(ref(GenModelRef));
 }
 let NewGenModel = undef, R = undef in seq{
  // create new model
  new(GenModel(NewGenModel) in nemf.resources);
  rename(NewGenModel, GenModelName+"_genmodel");
  update genmodel() = NewGenModel;
  // initialize GenModel using annotations
  new(relation(R,genmodel(),ecore()));
  call initialiseGenmodel(genmodel(),ecore());
  delete(R);
 }

 let GenType = nemf.packages.genmodel.GenModel in seq{
  // create GenModel equivalent of main EPackage
  call parseEcoreEntity(ecore(), genmodel(), GenType);
 }
 println("[INFO] >>> Ecore2Genmodel Transformation finished.");
}

//-------------------Type checking patterns--------------
pattern EcoreBoolean(EBoolean) = {nemf.ecore.datatypes.EBoolean(EBoolean);}
pattern EcoreString(EString) = {nemf.ecore.datatypes.EString(EString);}
pattern EcoreEnum(EEnum) = {nemf.ecore.datatypes.EEnum(EEnum);}
pattern EcoreEnumLiteral(EEnumLiteral) =
 {nemf.ecore.datatypes.EEnumLiteral(EEnumLiteral);}

pattern EPackage(EPackage) = {EPackage(EPackage);}
pattern ETypeParameter(ETypeParameter) = {ETypeParameter(ETypeParameter);}
pattern EEnumLiteral(EEnumLiteral) = {EEnumLiteral(EEnumLiteral);}
pattern EDataType(EDataType) = {EDataType(EDataType);}
pattern EClass(EClass) = {EClass(EClass);}
pattern EStructuralFeature(EStructuralFeature) =
 {EStructuralFeature(EStructuralFeature);}
pattern EOperation(EOperation) = {EOperation(EOperation);}
pattern EParameter(EParameter) = {EParameter(EParameter);}
pattern EEnum(EEnum) = {EEnum(EEnum);}
pattern EObject(EObject) = {EObject(EObject);}
pattern EReference(EReference) = {EReference(EReference);}

pattern GenPackage(GenPackage) = {GenPackage(GenPackage);}
```

8

```
pattern GenModel(GenModel) = {GenModel(GenModel);}
pattern GenClass(GenClass) = {GenClass(GenClass);}
pattern GenTypeParameter(GenTypeParameter) =
 {GenTypeParameter(GenTypeParameter);}
pattern GenEnumLiteral(GenEnumLiteral) = {GenEnumLiteral(GenEnumLiteral);}
pattern GenDataType(GenDataType) = {GenDataType(GenDataType);}
pattern GenFeature(GenFeature) = {GenFeature(GenFeature);}
pattern GenOperation(GenOperation) = {GenOperation(GenOperation);}
pattern GenParameter(GenParameter) = {GenParameter(GenParameter);}
pattern GenEnum(GenEnum) = {GenEnum(GenEnum);}

//-------------------------------------------------------

// initialize GenModel
rule initialiseGenmodel(in GenModel, in EcoreModel) = let R = undef in seq{

 // create default attributes required for a proper GenModel
 call parseAnnotation(GenModel,"copyrightFields",EcoreModel,"false","value");
 call parseAnnotation(GenModel,"complianceLevel",EcoreModel,
  nemf.packages.genmodel.GenJDKLevel.JDK60, "entity");
 call parseAnnotation(GenModel,"importerID",EcoreModel,
  "org.eclipse.emf.importer.ecore","value");

 // find modelname
 let Name =  nemf.packages.ecore.ENamedElement.name in
  try choose ModelName with
   find AttributeForType(EcoreModel,Name,ModelName) do
  let NewName = value(ModelName) in seq{
  update NewName = str.toUpperCase(str.substring(NewName,0,1))
   +str.substring(NewName,1);
  call parseAnnotation(GenModel,"modelName",EcoreModel,NewName,"value");
  // set foreign model reference
  let ForeignModel = undef in seq{
   new(nemf.ecore.datatypes.EString(ForeignModel) in GenModel);
   rename(ForeignModel, "foreignModel_"+value(ModelName));
   setValue(ForeignModel,value(ModelName));
   new(GenModel.foreignModel(R,GenModel,ForeignModel));
  }
 }

 call parseAnnotation(GenModel,"modelDirectory",EcoreModel,
  "/"+temp("pluginID")+"/src", "value");
 call parseAnnotation(GenModel,"modelPluginID",EcoreModel,
  temp("pluginID"), "value");
 // parse remaining annotations
 call parseAnnotationList(EcoreModel,GenModel);
}

// pattern for restricting datatypes
@localsearch
pattern EcoreDataType(EDataType) =
 {nemf.ecore.datatypes.EBoolean(EDataType);} or
  {nemf.ecore.datatypes.EString(EDataType);} or
 {nemf.ecore.datatypes.EEnum(EDataType);} or
 {nemf.ecore.datatypes.EEnumLiteral(EDataType);} or
 {nemf.ecore.datatypes.EInt(EDataType);}

// pattern for finding the type of an attribute
@localsearch
pattern AttributeForType(EcoreEntity,FeatureRel,Attribute) = {
 EModelElement(EcoreEntity);
 find MappedEcoreTypedEntity(EcoreEntity);
 nemf.ecore.EClass(EcoreType);
 instanceOf(EcoreEntity,EcoreType);
 nemf.ecore.EDataType(AttributeType);
 nemf.ecore.EClass.EAttribute(FeatureRel,EcoreType,AttributeType);
 entity(Attribute);
 find EcoreDataType(Attribute);
```

```
 instanceOf(Attribute,AttributeType);
 relation(Rel,EcoreEntity,Attribute);
 instanceOf(Rel,FeatureRel);
}

// create Ecore equivalent of EcoreEntity in GenModel
// under GenmodelParent which has type GenParentType
rule parseEcoreEntity(in EcoreEntity, in GenmodelParent, in GenParentType) =
 let GenmodelType = undef, NewGenmodelEntity = undef in seq{

 // find Ecore type from metamodel
 try choose EcoreType below nemf.packages.ecore with
  find EcoreType(EcoreEntity,EcoreType) do seq{
   // find GenModel type using mapping
  update GenmodelType = mapping(EcoreType);
 }
 else seq{
  println("[ERROR] Can't find Genmodel type!");
  fail;
 }

 if(GenmodelType != undef) let GenRefType = undef,
   PreviousRef = undef in seq{
  // create new entity
  new(entity(NewGenmodelEntity) in GenmodelParent);
  rename(NewGenmodelEntity,name(EcoreEntity));
  // set type of entity
  new(instanceOf(NewGenmodelEntity,GenmodelType));
  // create reference between GenModel and Ecore entity
  call createGenmodel2EcoreReference(NewGenmodelEntity,
   EcoreEntity,GenmodelType);
  // initialize Genmodel entity
  call initialiseGenmodelEntity(NewGenmodelEntity,EcoreEntity);

  // create GenModel entities for children
  forall EcoreChildEntity in EcoreEntity with
    find MappedEcoreTypedEntity(EcoreChildEntity) do seq{
   call parseEcoreEntity(EcoreChildEntity, NewGenmodelEntity, GenmodelType);
  }

  // handle external models (parsing is started from root EPackage)
  if(find EReference(EcoreEntity))
   try choose ExternalModel below nemf.resources with
     find ExternalEcoreEntityReference(EcoreEntity,ecore(),ExternalModel) do
   call parseEcoreEntity(ExternalModel, genmodel(), GenmodelType);

  // ordered relations are handled
  // for reference between same types
  if(GenmodelType == GenParentType)
   // find reference type
   try choose GenRefTypeT with
    find GenmodelEntitySelfReferenceType(GenmodelParent,NewGenmodelEntity,
      GenRefTypeT) do seq{
     update GenRefType = GenRefTypeT;
     // find last relation in the ordered list
     try choose PreviousRefT in GenmodelParent with
      find LastOrderedSelfRelationRef(GenParentType, GenmodelParent,
        GenRefType,PreviousRefT) do
       update PreviousRef = PreviousRefT;
   }
  // for reference between different types
  else // find reference type
    try choose GenRefTypeT with
      find GenmodelEntityReferenceType(GenmodelParent,
       NewGenmodelEntity,GenRefTypeT) do seq{
     update GenRefType = GenRefTypeT;
     // find last relation in the ordered list
     try choose PreviousRefT in GenmodelParent with
```

10

```
      find LastOrderedRelationRef(GenParentType,GenmodelType,
        GenmodelParent,GenRefType,PreviousRefT) do
       update PreviousRef = PreviousRefT;
   }
  if (GenRefType != undef) let R = undef in seq{
   // create reference between parent and child GenModel entities
   if (PreviousRef != undef) let RR = undef in seq{
    new(relation(R,GenmodelParent,NewGenmodelEntity));
    new(instanceOf(R,GenRefType));
    // create ordered relation
    new(nemf.ecore.EObject.orderedRelation.next(RR,PreviousRef,R));
   } else seq{
    new(relation(R,GenmodelParent,NewGenmodelEntity));
    new(instanceOf(R,GenRefType));
    }
  }
  // parse annotations for entity
  call parseAnnotationList(EcoreEntity,NewGenmodelEntity);
 }
}

// pattern for finding type for entity
@localsearch
pattern EcoreType(EcoreEntity, EcoreType) = {
 EModelElement(EcoreEntity);
 nemf.ecore.EClass(EcoreType) below nemf.packages.ecore;
 instanceOf(EcoreEntity,EcoreType);
 check(mapping(EcoreType) != undef);
}

// pattern for restricting entities to mapped types
@localsearch
pattern MappedEcoreTypedEntity(EcoreEntity) =
 {EPackage(EcoreEntity);} or
 {ETypeParameter(EcoreEntity);} or
 {EEnumLiteral(EcoreEntity);} or
 {EDataType(EcoreEntity);} or
 {EClass(EcoreEntity);} or
 {EStructuralFeature(EcoreEntity);} or
 {EOperation(EcoreEntity);} or
 {EParameter(EcoreEntity);} or
 {EEnum(EcoreEntity);
}

//pattern for checking already mapped Ecore entities
pattern MappedEcoreEntityInGenmodel(EcoreEntity) = {
 EModelElement(EcoreEntity);
 GenBase(GenmodelEntity);
 relation(R,GenmodelEntity,EcoreEntity);
}

// pattern for checking external model reference
@localsearch
pattern ExternalEcoreEntityReference(EReference, EcoreModel, ExternalModel)={
 EPackage(EcoreModel) below nemf.resources;
 EReference(EReference) below EcoreModel;
 EPackage(ExternalModel) below nemf.resources;
 EClass(ERefType) below ExternalModel;
 EReference.eReferenceType(R,EReference,ERefType);
 // check if the entity is not inside the model
 neg find MappedEcoreEntityInGenmodel(ExternalModel);
}

// pattern for finding reference type of GenModel entity and parent
// with the same entity type
pattern GenmodelEntitySelfReferenceType(GenParent,GenEntity,GenRefType) = {
 GenBase(GenParent);
 nemf.ecore.EClass(GenParentType);
```

```
 instanceOf(GenParent,GenParentType);
 GenBase(GenEntity) in GenParent;
 instanceOf(GenEntity,GenParentType);
 nemf.ecore.EClass.EReference(GenRefType,GenParentType,GenParentType);
 Boolean(True);
 check(True == datatypes.Boolean.true);
 nemf.ecore.EClass.EReference.containment(Containment,GenRefType,True);
}

// pattern for finding reference type of GenModel entity and parent
// with different entity types
pattern GenmodelEntityReferenceType(GenParent,GenEntity,GenRefType) = {
 GenBase(GenParent);
 GenBase(GenEntity) in GenParent;
 nemf.ecore.EClass(GenParentType);
 instanceOf(GenParent,GenParentType);
 nemf.ecore.EClass(GenmodelType);
 instanceOf(GenEntity,GenmodelType);
 nemf.ecore.EClass.EReference(GenRefType,GenParentType,GenmodelType);
 Boolean(True);
 check(True == datatypes.Boolean.true);
 nemf.ecore.EClass.EReference.containment(Containment,GenRefType,True);
}

// pattern for finding last relation among ordered relations of GenRefType
// (between same entity types)
pattern LastOrderedSelfRelationRef(GenParentType, GenParent,
  GenRefType,PreviousRef) = {
 GenBase(GenParent);
 nemf.ecore.EClass(GenParentType);
 nemf.ecore.EClass.EReference(GenRefType,GenParentType,GenParentType);
 instanceOf(GenParent,GenParentType);
 instanceOf(GenEntity,GenParentType);
 relation(PreviousRef,GenParent,GenEntity);
 instanceOf(PreviousRef,GenRefType);
 GenBase(GenEntity);
 find GenmodelEntitySelfReferenceType(GenParent,GenEntity,GenRefType);
 // match only if there is no next relation from the found reference
 neg pattern HasNextRelation(GenParent,GenEntity,PreviousRef,GenRefType) = {
  GenBase(GenParent);
  nemf.ecore.EClass(GenParentType);
  nemf.ecore.EClass.EReference(GenRefType,GenParentType,GenParentType);
  instanceOf(GenParent,GenParentType);
  instanceOf(GenEntity,GenParentType);
  relation(PreviousRef,GenParent,GenEntity);
  instanceOf(PreviousRef,GenRefType);
  GenBase(GenEntity);
  find GenmodelEntitySelfReferenceType(GenParent,GenEntity,GenRefType);
  // ---------------------------------------
  GenBase(OtherEntity);
  relation(R2,GenParent,OtherEntity);
  instanceOf(R2,GenRefType);
  nemf.ecore.EObject.orderedRelation.next(Rx,PreviousRef,R2);
 }
}

// pattern for finding last relation among ordered relations of GenRefType
// (between same entity types)
pattern LastOrderedRelationRef(GenParentType,GenmodelType, GenParent,
  GenRefType,PreviousRef) = {
 GenBase(GenParent);
 nemf.ecore.EClass.EReference(GenRefType,GenParentType,GenmodelType);
 instanceOf(GenParent,GenParentType);
 nemf.ecore.EClass(GenmodelType);
 nemf.ecore.EClass(GenParentType);
 instanceOf(GenEntity,GenmodelType);
 relation(PreviousRef,GenParent,GenEntity);
 instanceOf(PreviousRef,GenRefType);
```

12

```
 GenBase(GenEntity);// in GenParent;
 find GenmodelEntityReferenceType(GenParent,GenEntity,GenRefType);
 // match only if there is no next relation from the found reference
 neg pattern HasNextRelation(GenParent,GenEntity,PreviousRef,GenRefType) = {
  GenBase(GenParent);
  nemf.ecore.EClass.EReference(GenRefType,GenParentType,GenmodelType);
  instanceOf(GenParent,GenParentType);
  nemf.ecore.EClass(GenmodelType);
  nemf.ecore.EClass(GenParentType);
  instanceOf(GenEntity,GenmodelType);
  relation(PreviousRef,GenParent,GenEntity);
  instanceOf(PreviousRef,GenRefType);
  GenBase(GenEntity);
  find GenmodelEntityReferenceType(GenParent,GenEntity,GenRefType);
  // --------------------------------------
  GenBase(OtherEntity);
  relation(R2,GenParent,OtherEntity);
  instanceOf(R2,GenRefType);
  nemf.ecore.EObject.orderedRelation.next(Rx,PreviousRef,R2);
 }
}

// create cross-model reference from GenModel to Ecore
rule createGenmodel2EcoreReference(in GenmodelEntity,
  in EcoreEntity, in GenmodelType) = seq{
 // find reference type
 choose Gen2EcRefType with
  find Genmodel2EcoreReferenceType(GenmodelType,Gen2EcRefType) do
  let R = undef in seq{
   // create reference
   new(relation(R,GenmodelEntity,EcoreEntity));
   new(instanceOf(R,Gen2EcRefType));
  }
}

// pattern for finding reference type between GenModel and Ecore entities
@localsearch
pattern Genmodel2EcoreReferenceType(GenmodelType,Gen2EcRefType) = {
 nemf.ecore.EClass(GenmodelType) below nemf.packages.genmodel;
 nemf.ecore.EClass(EcoreType) below nemf.packages.ecore;
 nemf.ecore.EClass.EReference(Gen2EcRefType,GenmodelType,EcoreType);
}

// initialize GenModel entity (not GenModel typed) using annotations
rule initialiseGenmodelEntity(in GenmodelEntity, in EcoreEntity) = seq{

 // GenPackage-specific attributes needed for proper GenModel
 if(find GenPackage(GenmodelEntity)) seq{
  call parseAnnotation(GenmodelEntity,"disposableProviderFactory",
   EcoreEntity,"true", "value");
  let Name =  nemf.packages.ecore.ENamedElement.name in
  try choose ModelName with
    find AttributeForType(EcoreEntity,Name,ModelName) do seq{
   call parseAnnotation(GenmodelEntity,"prefix",EcoreEntity,
    value(ModelName), "value");
  }
 }
 // GenClass-specific attributes needed for proper GenModel
 else if(find GenClass(GenmodelEntity)) seq{
  let AbstractRel = nemf.packages.ecore.EClass.abstract in
  try choose Abstract with find AttributeForType(EcoreEntity,
    AbstractRel,Abstract) do seq{
   call parseAnnotation(GenmodelEntity,"image",EcoreEntity,
    toString(!(toBoolean(value(Abstract)))), "value");
  }
 }
 // GenEnum-specific attributes needed for proper GenModel
 else if(find GenEnum(GenmodelEntity)) seq{
```

```
 call parseAnnotation(GenmodelEntity,"typeSafeEnumCompatible",EcoreEntity,
  "false", "value");
}
// GenFeature-specific attributes needed for proper GenModel
else if(find GenFeature(GenmodelEntity)) let DefaultProperty = undef in seq{
 // EReference-specific attributes
 if(find EReference(EcoreEntity)) seq{
  let ContainerFeature = nemf.packages.ecore.EReference.container,
   ContainmentFeature = nemf.packages.ecore.EReference.containment,
   ChangeableFeature = nemf.packages.ecore.EStructuralFeature.changeable,
   Children = "false" in
  // find container, containment, children attribute types
  try choose Container with
   find AttributeForType(EcoreEntity,ContainerFeature,Container) do
   try choose Containment with
    find AttributeForType(EcoreEntity,ContainmentFeature,Containment) do
    try choose Changeable with
     find AttributeForType(EcoreEntity,ChangeableFeature,Changeable) do seq{
     // default property decided based on attributes
     if(!toBoolean(value(Container)) && !toBoolean(value(Containment))) seq{
      if(toBoolean(value(Changeable))) seq{
       update DefaultProperty =
        nemf.packages.genmodel.GenPropertyKind.Editable;
      }
      else seq{
       update DefaultProperty =
        nemf.packages.genmodel.GenPropertyKind.Readonly;
      }
     } else seq{
      update DefaultProperty = nemf.packages.genmodel.GenPropertyKind.None;
     }
     // children attribute created
     if(toBoolean(value(Containment)))
      call parseAnnotation(GenmodelEntity,"children",EcoreEntity,
       toString(toBoolean(value(Containment))), "value");
     else
      call parseAnnotation(GenmodelEntity,"children",
       EcoreEntity,"","value");
     try choose ChildrenT with
       find ChildrenAttribute(GenmodelEntity,ChildrenT) do seq{
       update Children = value(ChildrenT);
      }
     call parseAnnotation(GenmodelEntity,"createChild",
      EcoreEntity,toString((toBoolean(Children)
       && toBoolean(value(Changeable)))), "value");
     call parseAnnotation(GenmodelEntity,"notify",
      EcoreEntity,toString(toBoolean(Children)), "value");
     }
 }// otherwise (EAttribute)
  else seq{
  let ChangeableRel = nemf.packages.ecore.EStructuralFeature.changeable in
  // default property decided based on attributes
  try choose Changeable with
   find AttributeForType(EcoreEntity,ChangeableRel,Changeable) do seq{
   if(toBoolean(value(Changeable))) seq{
    update DefaultProperty =
     nemf.packages.genmodel.GenPropertyKind.Editable;
   }
   else seq{
    update DefaultProperty =
     nemf.packages.genmodel.GenPropertyKind.Readonly;
   }
   }
  call parseAnnotation(GenmodelEntity,"createChild",
   EcoreEntity,"false","value");
  call parseAnnotation(GenmodelEntity,"notify",EcoreEntity,"true","value");
 }
 if(find EReference(EcoreEntity)
```

14

```
    && DefaultProperty ==
     nemf.packages.genmodel.GenPropertyKind.Editable) seq{
   call parseAnnotation(GenmodelEntity,"propertySortChoices",
    EcoreEntity,"true", "value");
  } else call parseAnnotation(GenmodelEntity,"propertySortChoices",
   EcoreEntity,"false", "value");
  call parseAnnotation(GenmodelEntity,"property",
   EcoreEntity,DefaultProperty, "entity");
 }
}

// pattern to find children attribute for GenFeature entity
pattern ChildrenAttribute(GenEntity, ChildrenAttr) = {
 GenFeature(GenEntity);
 nemf.packages.genmodel.GenFeature.children(ChRel,GenEntity,ChildrenAttr);
 entity(ChildrenAttr);
 find EcoreDataType(ChildrenAttr);
}

// parse annotations for EcoreEntity, create attributes for GenModel entity
rule parseAnnotationList(in EcoreEntity, in GenmodelEntity) = seq{
 // forall annotation
 forall AnnotationKey below EcoreEntity with
   find UnmappedAnnotation(EcoreEntity,GenmodelEntity,AnnotationKey) do seq{
  // copy annotation value to attribute
  call parseAnnotation(GenmodelEntity,
   value(AnnotationKey),EcoreEntity,"","");
 }
}

// pattern to find annotations not yet handled
@localsearch
pattern UnmappedAnnotation(EcoreEntity,GenmodelEntity,AnnotationKey) = {
 EModelElement(EcoreEntity);
 GenBase(GenmodelEntity);
 relation(Re2g,GenmodelEntity,EcoreEntity);
 EModelElement.eAnnotations(Ra,EcoreEntity,EAnnotation);
 EAnnotation(EAnnotation);
 nemf.ecore.datatypes.EString(Source);
 EAnnotation.source(R,EAnnotation,Source);
 check(value(Source) == "emf.gen");
 EStringToStringMapEntry(Details);
 EAnnotation.details(R2,EAnnotation,Details);
 nemf.ecore.datatypes.EString(AnnotationKey);
 EStringToStringMapEntry.key(R3,Details,AnnotationKey);
 neg find ExistingAttributeForName(GenmodelEntity, AnnotationKey);
}

// pattern for checking existing attribute by its name
pattern ExistingAttributeForName(GenmodelEntity, AttributeName) = {
 EModelElement(EcoreEntity);
 relation(Re2g,GenmodelEntity,EcoreEntity);
 EModelElement.eAnnotations(Ra,EcoreEntity,EAnnotation);
 EAnnotation(EAnnotation);
 EStringToStringMapEntry(Details);
 EAnnotation.details(R2,EAnnotation,Details);
 nemf.ecore.datatypes.EString(AttributeName);
 EStringToStringMapEntry.key(R3,Details,AttributeName);
 GenBase(GenmodelEntity);
 nemf.ecore.EClass(GenmodelType);
 instanceOf(GenmodelEntity,GenmodelType);
 nemf.ecore.EDataType(AttributeType);
 nemf.ecore.EClass.EAttribute(AttributeRel,GenmodelType,AttributeType);
 String(Name);
 nemf.ecore.EClass.EStructuralFeature.name(Rn,AttributeRel,Name);
 check(value(Name)==value(AttributeName));
 entity(Attribute);
 instanceOf(Attribute,AttributeType);
```

```
 relation(Rel,GenmodelEntity,Attribute);
 instanceOf(Rel,AttributeRel);
}

// if annotation with given name exist, parse it, otherwise use default
rule parseAnnotation(in GenmodelEntity, in AnnotationKey,
  in EcoreEntity, in Default, in DefaultType) =
 let AnnValue = undef, DefUsed = false in seq{

 // find value from annotation
 try choose AnnValueT with
  find AnnotationValue(EcoreEntity, AnnotationKey, AnnValueT) do seq{
  update AnnValue = value(AnnValueT);
 // use default value otherwise
  } else seq{
  if (Default != "") seq{
   update AnnValue = Default;
   update DefUsed = true;
  }
 }

 if(AnnValue != undef) seq{
  // find attribute type and relation
  try choose AttributeType, AttributeRel with
   find AttributeTypeForName(GenmodelEntity,AnnotationKey,AttributeType,
    AttributeRel) do let Attribute = undef, AttrR = undef in seq{
    // create relation to default entity
    if (DefUsed == true && DefaultType == "entity") seq{
     new(relation(AttrR,GenmodelEntity,Default));
    new(instanceOf(AttrR,AttributeRel));
     }else let Value = str.trim(AnnValue), Rest = str.trim(AnnValue) in
    // handle EReference.many (iterate always finds Many)
    iterate choose Many in datatypes.Boolean with
     find IsAttributeRelMany(AttributeRel,Many) do
     // update params
      let Start = 0, End = str.indexOf(Rest,",") in seq{
     // if many relation and has more values, parse single value
     if(Many == datatypes.Boolean.true && End > Start) seq{
      update Value = str.substring(Rest,Start,End);
     } else seq{
      update Value = Rest;
     }
     // checking value for type safety (Boolean, Enum)
     let Result = undef, Target = undef in seq{
      // handle EBoolean true/false, EEnum values, reference to enums
     call checkAttributeTypeInAnnotationValue(Value,
       AttributeType,Result,Target);
      if(Result == "ok") seq{
       if(Target == undef) seq{
       // create attribute
        new(entity(Attribute) in GenmodelEntity);
       new(instanceOf(Attribute,AttributeType));
       rename(Attribute,AnnotationKey);
       setValue(Attribute,Value);
       // create Attribute relation
       new(relation(AttrR,GenmodelEntity,Attribute));
       new(instanceOf(AttrR,AttributeRel));
       } else if(find EcoreEnumLiteral(Target)) seq{
        // create Attribute relation
       new(relation(AttrR,GenmodelEntity,Target));
       new(instanceOf(AttrR,AttributeRel));
      }
     }// return fault
      else if(str.startsWith(Result,"fault_")) seq{
      println("[ERROR] Annotation Value is wrong: "
       + str.substring(Result,6));
     }
     }
```

16

```
    // update Rest if there is more of it (or comma is the last char)
    if(End > 0 && End < str.length(Rest)-1)
     update Rest = str.substring(Rest,End+1);
    // otherwise exit loop
    else fail;
    }
   }
  else seq{
   println("[Warning] No such attribute (" + AnnotationKey
    + ") in genmodel for (" + GenmodelEntity + ")");
  }
 }
}

// pattern for finding annotation value for given key
@localsearch
pattern AnnotationValue(EcoreEntity, AnnotationKey, AnnotationValue) = {
 EAnnotation(EAnnotation) in EcoreEntity;
 nemf.ecore.datatypes.EString(Source) in EAnnotation;
 EAnnotation.source(R,EAnnotation,Source);
 check(value(Source) == "emf.gen");
 EStringToStringMapEntry(Details) in EAnnotation;
 EAnnotation.details(R2,EAnnotation,Details);
 nemf.ecore.datatypes.EString(Key) in Details;
 EStringToStringMapEntry.key(R3,Details,Key);
 check(value(Key) == AnnotationKey);
 nemf.ecore.datatypes.EString(AnnotationValue) in Details;
 EStringToStringMapEntry.value(R4,Details,AnnotationValue);
}

// pattern for retrieving "many" value of relation
pattern IsAttributeRelMany(AttributeRel,Many) = {
 nemf.ecore.EClass(EcoreType);
 nemf.ecore.EDataType(AttributeType);
 nemf.ecore.EClass.EAttribute(AttributeRel,EcoreType,AttributeType);
 Boolean(Many);
 nemf.ecore.EClass.EStructuralFeature.many(ManyRel,AttributeRel,Many);
}

// pattern for finding attribute type and relation by name
@localsearch
pattern AttributeTypeForName(GenmodelEntity, AttributeName,
  AttributeType, AttributeRel) = {
 GenBase(GenmodelEntity);
 nemf.ecore.EClass(GenmodelType);
 instanceOf(GenmodelEntity,GenmodelType);
 nemf.ecore.EDataType(AttributeType);
 nemf.ecore.EClass.EAttribute(AttributeRel,GenmodelType,AttributeType);
 String(Name) in GenmodelType;
 nemf.ecore.EClass.EStructuralFeature.name(R,AttributeRel,Name);
 check(value(Name)==AttributeName);
}

// Result is "ok" if type is in order, otherwise another string starting
// with "fault_" followed by the reason.
// Target is an EEnumLiteral entity
// if such an attributetype and value is given
rule checkAttributeTypeInAnnotationValue(in Value, in AttributeType,
  out Result, out Target) = seq{
 update Result = "ok";
 update Target = undef;
 // handle EBoolean true/false
 if(find EcoreBoolean(AttributeType)) seq{
  if(Value != "true" && Value != "false") seq{
   update Result = "fault_Not Boolean value: "+Value+", expected true/false";
   //update Target = undef;
  }
 }
```

```
 // handle EEnum values
 if(find EcoreEnum(AttributeType)) seq{
  // handle reference to enums
  try choose EnumValue in AttributeType with
    find EnumLiteralInEnumType(EnumValue,Value,AttributeType) do seq{
   //update Result = "ok";
   update Target = EnumValue;
  } else seq{
   //update Target = undef;
   update Result = "fault_Value "+Value+" is not "
    + name(AttributeType) + ", expected |";
   forall EnumValue in AttributeType with
     find EnumLiteralsInEnumType(EnumValue,AttributeType) do seq{
    update Result = Result + name(EnumValue) + "|";
   }
   update Result = Result;
  }
 }
}

// pattern for finding EnumLiteral type in Enum by its name
@localsearch
pattern EnumLiteralInEnumType(EnumValue,Value,AttributeType) = {
 nemf.ecore.datatypes.EEnum(AttributeType);
 nemf.ecore.datatypes.EEnumLiteral(EnumValue);
 instanceOf(EnumValue,AttributeType);
 check(value(EnumValue) == Value);
}

// pattern for finding EnumLiterals for an Enum
pattern EnumLiteralsInEnumType(EnumValue,AttributeType) = {
 nemf.ecore.datatypes.EEnum(AttributeType);
 nemf.ecore.datatypes.EEnumLiteral(EnumValue);
 instanceOf(EnumValue,AttributeType);
}
}
```

Listing 1.1. Transformation code