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 JGraLab1 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) programatically2 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.

1http://jgralab.uni-koblenz.de
2It 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 transformation constructs

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

```java
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 VertexClasses 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.

```java
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.

```java
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.

```java
for (String[] s : new String[][] { { "InitialNode", "initial" }, { "ForkNode", "fork" },
    { "JoinNode", "join" }, { "DecisionNode", "junction" } }) {
    createVertexClass(s[0],
”from ps : V{" + s[1] + "} 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.

```java
for (String[] s : new String[][] { { "Activity", "ActivityGraph" },
    { "ActivityPartition", "Partition" }, { "ActivityFinalNode", "FinalState" },
    { "OpaqueAction", "ActionState" }, { "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.

```java
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 instantiated, 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.

```java
createEdgeClass("ActivityContainsGroup",
               new IncidenceClassSpec(vc("Activity")),
               new IncidenceClassSpec(vc("ActivityPartition"), AggregationKind.COMPOSITE),
               "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 component 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.

```java
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 establishing 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.

```java
addSubClasses(activityNode, vc("OpaqueAction"), vc("InitialNode"),
vc("ActivityFinalNode"), vc("DecisionNode"), vc("JoinNode"),
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 imgActivityNode and archActivityNode 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.

```java
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.

```java
createAttribute(new AttributeSpec(modelElement, "name", getStringDomain()),
"from me : difference(keySet(img_ModelElement), keySet(img_ActivityEdge)) "
+ "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 archetypes of target graph ModelElements excluding the archetypes of ActivityEdges are iterated. 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

---

3The GReQL function keySet() returns the set of keys of a map, that is the domain of the imgModelElement 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 transferred to the ActionState v6 via the two Transition vertices v22 and v23 and their connecting ComesFrom (e25, e26) 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.
Preceding Shared Part. The metamodel of both tasks is identical, so the creation of meta-
model 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.

These two operation invocations create the ObjectFlow vertex class and the HasObject compo-
sition edge class according to the target schema (figure). 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 when transformed according to the object flow extension task is shown in figure. 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.

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 instantiate-
eVertices() and instantiateEdges(), which only work on the instance level.

*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](image)

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.

```scala
 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.

```java
VertexClass controlFlow = createVertexClass("ControlFlow", "from t : V(Transition) + " with isEmpty(t -->\{ComesFrom, GoesTo\} & \{ObjectFlowState\}) + " reportSet t, t, end"; VertexClass activityEdge = createAbstractVertexClass("ActivityEdge"); 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 $\text{img}_{\text{ActivityEdge}}$ 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 straightforward.

```java
createEdgeClass("ComesFrom", new IncidenceClassSpec(activityEdge), new IncidenceClassSpec(vc("ActivityNode"), 1, 1), "from t : keySet(img_ActivityEdge) + " reportSet t, t, theElement(t[0] -->\{ComesFrom\}) end");
createEdgeClass("GoesTo", new IncidenceClassSpec(activityEdge), new IncidenceClassSpec(vc("ActivityNode"), 1, 1), "from t : keySet(img_ActivityEdge) + " 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 $\text{img}_{\text{ActivityEdge}}$ 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 concise and specify the correlation between elements in a very natural, declarative manner.
A The TGraph Approach

In the TGraph 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”, 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
Each TGraph schema itself conforms to the grUML metaschema, which is the metamodel of the TGraph approach. The metaschema is a valid schema describing itself. Its core is depicted in figure 9.

![Diagram](image_url)

**Figure 8:** The UML 1.4 activity diagram source schema

**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,

---

[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 (\( \mid \)), 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 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:

| (v7: ObjectFlowState, Placed Order) | (v17: ActionState, Restock) |
| (v12: ActionState, Pay) | (v14: ActionState, Collect order) |
| (v6: ActionState, Take order) | (v8: ActionState, Fill Order) |
| (v9: ObjectFlowState, Entered Order) | (v19: FinalState, Finished) |
| (v13: ActionState, Deliver order) | (v11: ObjectFlowState, Filled Order) |
| (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


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.

```java
public class Activity1ToActivity2 extends Transformation {
    public enum Task { CORE, OBJECT_FLOW_EXTENSION }
    private Task task;
    public void setTask(Task task) { this.task = task; }
    public Activity1ToActivity2(Context c) { super(c); }

    @Override protected void transform() {
        // This array contains pairs {NewType, OldType}.
        // Schema Level: For each OldType vertex class, a new vertex class with qualified
        // name NewType is created in the target schema.
        // Instance Level: For each source graph OldType vertex, a new NewType vertex is
        // created in the target graph.
        for (String[] s : new String[][] { { "Activity", "ActivityGraph" },
            { "ActivityPartition", "Partition" }, { "ActivityFinalNode", "FinalState" },
            { "OpaqueExpression", "Guard" } }) {
            createVertexClass(s[0], "V{" + s[1] + "}");
        }

        // Initial-, Fork-, Join-, DecisionNodes are all Pseudostates in UML1.4, only
        // distinguishable by their kind attribute. So this array contains pairs {NewType,
        // kindAttrValue}.
        // Schema Level: For each array element, create a vertex class with qualified name
        // NewType in the target metamodel.
        // Instance Level: For each source model Pseudostate with kind = kindAttrValue,
        // create one target model vertex of type NewType.
        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 ");
        }

        // Schema Level: Create an abstract vertex class ActivityNode in the target schema.
        // Instance Level: Abstract classes don’t have instances, so this operation doesn’t
        // affect the instance level.
        VertexClass activityNode = createAbstractVertexClass("ActivityNode");

        // Schema Level: Make ActivityNode the superclass of all the following 6 vertex
        // classes that were already created by the previous operation calls. The method
        // vc(String) simply retrieves the target metamodel vertex class with the given
        // qualified name.
```
addSubClasses(activityNode, vc("OpaqueAction"), vc("InitialState"),
    vc("ActivityFinalNode"), vc("DecisionNode"), vc("JoinNode"),
    vc("ForkNode"), vc("ObjectNode"));

// Schema Level: Create the vertex class ObjectFlow in the target schema.
// Instance Level: No effect.

// Schema Level: Create a new edge class HasObject with composition semantics from
// ObjectFlow to ObjectNode in the target schema. ObjectFlow is the whole and
// ObjectNode is on the part side.
// Instance Level: Instances are only needed in the extension task, but not in the
// core task.

VertexClass objectFlow = createVertexClass("ObjectFlow");

EdgeClass hasObject = createEdgeClass("HasObject",
    new IncidenceClassSpec(objectFlow, 0, 1),
    new IncidenceClassSpec(vc("ObjectNode"), 0, 1, AggregationKind.COMPOSITE));

switch (task) {
    case CORE:
        // Schema Level: already done above, so nothing to do here.
        // Instance Level: For each Transition vertex t, which goes to or comes from an
        // ObjectFlowState, create one ObjectFlow vertex in the target graph. As
        // archetype, we use a tuple where both components are the transition t, because
        // this makes the structure uniform to the OBJECT_FLOW_EXTENSION task.
        instantiateVertices(objectFlow,
            "from t : V\{Transition\} "
            + "with not isEmpty(t --->{GoesTo, ComesFrom} & {ObjectFlowState}) "
            + "reportSet t, t end ");
        break;
    case OBJECT_FLOW_EXTENSION:
        // Schema Level: already done above, so nothing to do here.
        // Instance Level: For each pair of Transition vertices (t1, t2), which build up
        // a source model structure like
        // t1 --->{GoesTo} & {ObjectFlowState} \longrightarrow {ComesFrom} t2
        // create one ObjectFlow vertex in the target graph. Use the tuple (t1, t2) as
        // archetype for the new object flow.
        instantiateVertices(objectFlow,
            "from t1 : V\{Transition\}, t2 : V\{Transition\} 
            + "with t1 --->{GoesTo} & {ObjectFlowState} \longrightarrow {ComesFrom} t2 
            + "reportSet t1, t2 end ");

        // Schema Level: already done above, so nothing to do here.
        // Instance Level: For each Transition tuple t in the domain of img_ObjectFlow
// create one HasObject edge. It starts at the image of the tuple t which is an
// ObjectFlow, and it ends at the image of the tuple's first Transition's
// GoesTo-target, which is some ObjectFlowState for which we've already created
// an ObjectNode.
instantiateEdges(hasObject,
  "from t : keySet(img_ObjectFlow) " +
  "reportSet t , t , theElement(t[0] -->(GoesTo)) end " ;
break;
default:
  throw new GReTLException(context, "Unknown task " + task + "!" );
// That was the whole task-specific part. Due to the uniform selection of archetypes,
// the rest can be handled uniformly.

// Schema Level: Create a vertex class ControlFlow in the target schema.

// Instance Level: For each source model Transition which doesn’t start or end at an
// ObjectFlowState create one ControlFlow vertex in the target graph. Nevertheless,
// we use transition tuples as archetypes, so that img_ControlFlow’s domain is
// structurally equal to the domain of img_ObjectFlow, that is (Transition x
// Transition).
VertexClass controlFlow = createVertexClass("ControlFlow",
  "from t : V{Transition} "
  + "with isEmpty(t -->{ComesFrom, GoesTo} & {ObjectFlowState}) "
  + "reportSet t , t end end " ;
// Schema Level: Create the abstract vertex class ActivityEdge.

// Instance Level: No effect.
VertexClass activityEdge = createAbstractVertexClass("ActivityEdge");

// Schema Level: Make ControlFlow and ObjectFlow specializations of ActivityEdge.
// Instance Level: No effect.
addSubClasses(activityEdge , controlFlow , objectFlow );

// Schema Level: Creates the composition edge class HasGuard.
// Instance Level: For each HasGuard edge in the source model create a HasGuard edge
// in the target model. The start vertex is the image of a tuple
// (connectedTransition , connectedTransition), and that’s exactly the archetype we
// chose for ActivityEdges. The end vertex is the image of the edge’s end vertex,
// which is a Guard. Above, we created one OpaqueExpression for each Guard.
createEdgeClass("HasGuard",
  new IncidenceClassSpec(activityEdge , 0, 1),
  new IncidenceClassSpec(vc("OpaqueExpression") , 0, 1, AggregationKind.COMPOSITE),
  "from e : E[HasGuard] "
  + "reportSet e , tup(startVertex(e) , startVertex(e)) , endVertex(e) end ");
// Schema Level: Create the body attribute of type String for the vertex class
// OpaqueExpression.
// Instance Level: Simply set the value according the Guard g’s source model
// condition body, which is some BooleanExpression.
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",
  ...
```java
new IncidenceClassSpec(vc("ActivityPartition"), 0, 1),
new IncidenceClassSpec(vc("ActivityNode"), AggregationKind.COMPOSITE),
"from e : E{ContainsElement}"
+ "with containsKey(img_ActivityNode, endVertex(e))"
+ "reportSet endVertex(e), startVertex(e), endVertex(e) end")

// Schema Level: Create a composition edge class ContainsEdge.
// ActivityPartition (0,1) -> ContainsEdge -> ActivityEdge
// Instance Level: The source model has no ContainsElement edges from Partitions to Transitions, so we add some heuristics here. Each ActivityEdge will be contained in all ActivityPartitions of its source and target ActivityNode.
createEdgeClass("ContainsEdge",
new IncidenceClassSpec(vc("ActivityPartition"), 0, 1),
new IncidenceClassSpec(activityEdge, AggregationKind.COMPOSITE),
"from t : keySet(img_ActivityEdge).
  p : flatten(from s : union(t[0]--->{ComesFrom}, t[1]--->{GoesTo})
    reportSet s <--{ContainsElement} end"
  + "reportSet tup(t, p, p, t end")

// Schema Level: Create a composition edge class ActivityContainsGroup.
// Activity (1,1) -> ActivityContainsGroup -> ActivityPartition
// Instance Level: For each source model HasPartition edge, create an ActivityContainsGroup edge starting at the image of this edge's start vertex (an ActivityGraph) to the image of this edge's end vertex (a Partition).
createEdgeClass("ActivityContainsGroup",
new IncidenceClassSpec(vc("Activity")),
new IncidenceClassSpec(vc("ActivityPartition"), AggregationKind.COMPOSITE),
"from e : E{HasPartition}"
+ "reportSet e, startVertex(e), endVertex(e) end");

// Schema Level: Create a composition edge class ActivityContainsNode.
// Activity (0,1) -> ActivityContainsNode -> ActivityNode
// Instance Level: For each ActivityNode archetype (some StateVertex) which is connected to no ContainsElement edge, create one ActivityContainsNode from the image of the nearest ActivityGraph containing this archetype to its own image.
// The constraint ensures that no ActivityContainsNode edges are created for ActivityNodes which are already contained in an ActivityPartition which in turn is contained in an Activity.
createEdgeClass("ActivityContainsNode",
new IncidenceClassSpec(vc("Activity"), 0, 1),
new IncidenceClassSpec(activityNode, AggregationKind.COMPOSITE),
"from a : keySet(img_ActivityNode)"
+ "with degree{ContainsElement}(a) = 0"
+ "reportSet a, theElement(a ---{a} & [ActivityGraph]), a end");

// Schema Level: Create a composition edge class ActivityContainsEdge.
// Activity (0,1) -> ActivityContainsEdge -> ActivityEdge
// Instance Level: For each ActivityEdge archetype (those are (Transition,
```
// Transition) tuples), for which no ContainsEdge has already been created, create
// one ActivityContainsEdge from the image of the nearest ActivityGraph containing
// the transition \([0]\) to the image of the tuple itself, which is an ActivityEdge.

// Again, the condition in the with-clause ensures that no ActivityContainsEdge edges
// are created for ActivityEdges which are already contained in an ActivityPartition
// which in turn is contained in an Activity.

createEdgeClass("ActivityContainsEdge",
    new IncidenceClassSpec(vc("Activity"), 0, 1),
    new IncidenceClassSpec(activityEdge, AggregationKind.COMPOSITE),
    "from t : keySet(img_ActivityEdge)
        + "with isEmpty(from x : keySet(img_ContainsEdge)
          + "with x[0] = t
          + "reportSet x end"
        + "reportSet t, theElement(t[0] -->> & {ActivityGraph}), t end") ;

// Schema Level: Create the abstract vertex class ModelElement.
// Instance Level: No effect.
VertexClass modelElement = createAbstractVertexClass("ModelElement") ;

// Schema Level: Make ModelElement the superclass of ActivityEdge, Activity,
// ActivityPartition, ActivityNode and ObjectNode.
// Instance Level: No effect.
addSubClasses(modelElement, activityEdge, vc("Activity"),
    vc("ActivityPartition"), vc("ActivityNode")) ;

// Schema Level: Create a name attribute of type String at the ModelElement vertex
// class.
// Instance Level: For each target ModelElement excluding ActivityEdges, set the
// value of the attribute according to the value of the archetype.
Attribute name = createAttribute(
    new AttributeSpec(modelElement, "name", getStringDomain()),
    "from me : difference(keySet(img_ModelElement), "
    + "keySet(img_ActivityEdge) "
    + "reportMap me, me. name end"
) ;

// Schema Level: No effect.
// Instance Level: For each ActivityEdge archetype, set the name of the image
// according their predecessor/successor−ActivityNode. So if an activity edge starts
// at an "A" action and goes to a "B" action, the ActivityEdge 's name will be "A -->
// B". The GRel operator ++ is used to concatenate two strings.

public static void main(String[] args) throws GraphIOException {
    // Create a new Context object holding the state of the transformation. The name of
    // the target schema to be created by the transformation is
    // de.uni_koblenz.umi2.ActivitySchema, and it will define the graph class
    // ActivityGraph.
Context c = new Context("de.uni_koblenz.uml2.ActivitySchema", "ActivityGraph");

// Set the source model for the transformation. The source graph is created
// programmatically by the CreateUML1ActivityGraph class in the util package. You
// might want to have a look.
\n\nc.setSourceGraph(CreateUML1ActivityGraph.getUml1ADGraph());

// Instantiate our transformation.
Activity1ToActivity2 act1ToAct2 = new Activity1ToActivity2(c);

// Run the transformation once with CORE and the second time with
// OBJECT_FLOW_EXTENSION task. It will create the UML2 target metamodel/schema in
// the same time migrate the UML1.4 model to a target model conforming to the just
// created UML2.2 schema.
for (Task task : new Task[]{Task.CORE, Task.OBJECT_FLOW_EXTENSION}) {
    // Reset the context (that won’t forget the target schema, so that the
    // transformation will reuse the schema created in the first run).
    c.reset(false);
    // Set the current task.
    act1ToAct2.setTask(task);
    // Execute the transformation.
    act1ToAct2.execute();
    // Retrieve the target model.
    Graph targetGraph = c.getTargetGraph();
    // Save the target model in the native TG format.
    GraphIO.saveGraphToFile("uml2model_" + task.toString().toLowerCase() + ".tg",
        targetGraph, new ProgressFunctionImpl());
    // Also save a visualization of the target model in the GraphViz DOT format. You
    // can view that using GraphViz’ dotty program.
    Tg2Dot.printGraphAsDot(targetGraph, false, "uml2model_" + task.toString()
        .toLowerCase() + ".dot", targetGraph.getSchema()
        .getAttributeClass("ComesFrom").getM1Class());
}