Solving the TTC 2011 Reengineering Case with GReTL

Dipl.-Inform. Tassilo Horn

horn@uni-koblenz.de
Institute for Software Technology
University Koblenz-Landau, Campus Koblenz

This paper discusses the GReTL reference solution of the TTC 2011 Reengineering case [5]. Given a Java syntax graph, a simple state machine model has to be extracted. The submitted solution covers both the *core task* and the two *extension tasks*.

1   Introduction

GReTL (Graph Repository Transformation Language, [6]) is the operational transformation language of the TGraph technological space [2]. Models are represented as typed, directed, ordered, and attributed graphs. GReTL uses GReQL (Graph Repository Query Language, [1]) for its querying part.

In contrast to most other transformation languages, GReTL transformations usually construct the target metamodel (*schema*) on their own, thereby specifying one graph conforming to this new schema as target graph of the transformation. For this purpose, GReTL provides a slim set of four transformation operations, which are derived from the metametamodel of the TGraph technological space (the *GraphUML metaschema*). There is an operation *CreateVertexClass*, which creates a new node type (*VertexClass*) in the target schema and a set of vertices of this new type in the target graph. Likewise, there is an operation *CreateEdgeClass*, which creates a new edge type (*EdgeClass*) in the target schema and a set of edges of this new type in the target graph. Since schemas allow for multiple inheritance between vertex as well as edge classes, there is an operation *AddSubClass* to create specialization relationships in the target schema. Finally, there is an operation *CreateAttribute*, which creates a new attribute for a vertex or edge class and which assigns values to the elements for which that new attribute is defined. The vertices and edges that have to be created in the target graph as well as the function assigning values are specified in terms of queries on the transformation's source graph.

2   Task Solutions

In this section, all tasks are discussed in sequence, and the GReTL operations and GReQL queries are explained when they come along. The solution can be run on the SHARE image [4].

2.1   The Core Task

The core task is responsible for creating States and Transitions without setting the attributes of the latter. Because the JaMoPP metamodel [3] splits its types into various packages, we import the packages from which elements are used, so that we can refer to these types without having to qualify them.

```
1  import classifiers.*; import types.*;    import members.*;
2  import references.*;  import statements.*; import modifiers.*;
```

The first task is the creation of State vertices. As explained in the task description, there is an abstract Java class named State, and all concrete subclasses can be considered states. At first, we bind abstract State class to a variable abstractStateClass, so that we can easily refer to it in the transformation.

```
3   abstractStateClass := theElement(from c: V{Class}
4                                     with c.name = "State" reportSet c end);
```

The *from-with-reportSet* expression calculates the set of classes which are named "State". The function theElement() extracts the single element of a collection consisting of only one element and throws an exception if the collection's size is not one. This expresses the assumption that there is exactly one state class. Finally, this class is assigned to the variable abstractStateClass.

The first transformation operation invoked is CreateVertexClass. It creates a new vertex class State in the target schema. The query following the arrow symbol is evaluated on the source graph and has to result in a set. For each member of this set (*archetype*), a new vertex (*image*) of the just created type is instantiated in the target graph. The mappings from archetypes to target graph images are automatically saved in a function corresponding to the target metamodel vertex class (img_{State}).

```
5   CreateVertexClass State
6     <== from c: {Class} & (<>--{extends} <>--{classifierReferences} -->{target})+
7                            abstractStateClass
8         with isEmpty(c <>--{annotationsAndModifiers} & {Abstract})
9         reportSet c end;
```

The query specifies a set of Class vertices. The variable c iterates over Class vertices, for which a path to the vertex abstractStateClass exists. The structure of this path is specified using a *regular path expression* [1]. First, a containment edge with role name extends at the far end has to be traversed, followed by another containment edge with role name classifierReferences, followed by a forward edge with role name target. This is exactly how subclasses relate to their superclass. The + specifies a one-or-many iteration. Thus, c is bound not only to direct subclasses of abstractStateClass, but also to indirect ones. The with part ensures that c is not abstract, i.e., it must not reference an Abstract vertex using an edge with containment semantics and far end role name annotationsAndModifiers. For any non-abstract class that extends the abstract state class either directly or indirectly, a new target graph State vertex is created. The mappings from classes to states are stored in a function img_State, which can be used in following operation calls for navigating between archetypes and images.

The next operation creates the name attribute of type String for the State vertex class, and it sets the attribute values for the vertices created by the last operation call.

```
10   CreateAttribute State.name : String
11     <== from c: keySet(img_State) reportMap c -> c.name end;
```

The query of the CreateAttribute operation has to result in a map assigning values of the attribute's type to archetyps. Here, the map assigns class names to State archetypes, so the state names are set to the names of the classes they were created for.

The CreateEdgeClass operation is used to create a new edge type Transition in the target metamodel defined between State vertices with the given role names and default multiplicities (0,*). The query has to result in a set of triples. In each triple, the first component specifies the archetype of the new edge to be created. The second and third component specify the archetypes of the start and end vertices. For each archetype, a new edge of the just created type is created in the target graph, starting at the vertex that is the image of the second component and ending at the vertex that is image of the third component.

```
12   CreateEdgeClass Transition from State role src to State role dst
```

```
13    <== from c1, c2: keySet(img_State),
14           callingMethod: c1 <>--{members} & {Method},
15           call: callingMethod <>--+ & {MethodCall}
16       with call -->{target} instanceMethod
17         and not isEmpty(call <>--{next} & {MethodCall} -->{target}
18                        & {Method @ thisVertex.name = "activate"})
19       reportSet tup(c1, callingMethod, c2, instanceMethod), c1, c2 end
20       where instanceMethod := theElement(c2 <>--{members}
21                        & {Method @ thisVertex.name = "Instance"});
```

In the query, c_1 and c_2 iterate over State archetypes, i.e., source graph Class vertices extending the abstract state class. The variable callingMethod is bound to all methods of the class bound to c_1 one after the other. In turn, call is bound to every MethodCall occuring somewhere in callingMethod's body using the regular path expression <>--+, which calculates all method call vertices reachable from callingMethod by traversing edges with containment semantics one or many times. The variable instanceMethod is bound to the singleton Instance() Method of c_2 using a where binding. The predicates in the with part ensurne that the method call call indeed invokes the instanceMethod and that on the object returned by the call, the activate() method is invoked. For each variable combination fulfilling the predicates, a triple is reported. The first component, i.e., the archetype for a new Transition edge, is a tuple containing the currently active state class c_1, its method that contains the activation call (callingMethod), the new state class c_2, and its instanceMethod. The archetype of the new transition's start state is c_1 and the transition leads to the state that is the image of c_2.

This is all the core task requires. In the remainder, the extension solutions are discussed.

2.2   Extension 1: Triggers

With respect to triggers, four cases have to be distinguished: (1) If the transition occurs in a method except for run(), then that method's name is the trigger. (2) If it occurs in a switch statement in the run() method, then the corresponding case's enum constant name is the trigger. (3) If it occurs in a catch block, then the caught exception's type name is the trigger. (4) Else, the trigger should be set to the string "−−". The four situations are covered by different operation calls.

Non-run() methods and default value.   The CreateAttribute operation is used to create the trigger attribute of type String for the Transition edge class. The string "−−" is chosen as default value, which handles the fourth case above.

```
22   CreateAttribute Transition.trigger : String = '"−−"'
23    <== from t: keySet(img_Transition)
24       with t[1].name <> "run"
25       reportMap t -> t[1].name end;
```

The query returns a map that assigns to every Transition archetype whose second component's name is not "run" the value of its name. When looking at the CreateEdgeClass call for Transition in the core task, the second component of Transition archetypes (t[1]) is exactly the method in which the activation of the new state occured.

Switch statements.   Because the trigger attribute has already been created by the previous CreateAttribute call, the SetAttributes operation is used, which only works on the instance level and requires an existing attribute given by its qualified name (Transition.trigger).

```
26    SetAttributes  Transition.trigger
27       <== from  t:  keySet(img_Transition),
28              case:  t[1] <>—+ & {Switch}<>--{cases},
29              cond:  case <>--{condition} -->{target} & {EnumConstant}
30           with  t[1].name = "run"  and  case <>—+ & {MethodCall} -->{target}  t[3]
31           reportMap  t -> cond.name  end;
```

The query iterates over Transition archetypes (4-tuples) using the variable t. case is bound to all Case vertices reachable by diving into the body of the activating method referenced by t[1], reaching a Switch vertex, and selecting its Case vertices one after the other. In turn, cond is bound to the EnumConstant that is the condition of the case. The predicates in the with part ensure that the activation of the next state occurs inside the run() method, and that the call of the Instance() method (the fourth component in the transition archetype tuples) indeed occurs in the body of case. The map assigns the names of the EnumConstant used in the case of the switch statement to Transition archetypes.

Catch blocks.   Again, the SetAttributes operation is used.

```
32    SetAttributes  Transition.trigger
33       <== from  t:  keySet(img_Transition),  catch:  t[1] <>—+ & {CatchBlock}
34           with  t[1].name = "run"  and  catch <>—+ & {MethodCall} -->{target}  t[3]
35           reportMap  t -> theElement(catch -->{parameter} -->{typeReference}
36                                            -->{classifierReferences} -->{target}).name  end;
```

The query iterates over Transition archetypes using the variable t. From the method containing the activation call (t[1]), all CatchBlock vertices contained in its body are iterated. The with part ensures the activation is inside the run() method and the activation call occurs inside the catch block. For all combinations of t and catch where the predicates hold, the name of the caught exception is the trigger.

2.3   Extension 2: Actions

The second extension task deals with setting the action attribute of Transition edges. The value of this attribute is the name of the enumeration constant provided as argument to a send() method call appearing in the same block as the activation of the next state. If there is no such call, the attribute should be set to "−−". The CreateAttribute operation is used to create the new action attribute of type String for the edge class Transition with the default value "−−".

```
37    CreateAttribute  Transition.action  :  String  = '"−−"'
38       <== from  t:  keySet(img_Transition),
39              container:  t[1] <>--* & {StatementListContainer},
40              sendCall:  container <>--{statements} <>--{expression} & {MethodCall}
41           with  not isEmpty(sendCall -->{target} & {Method @ thisVertex.name = "send"})
42             and  container <>--{statements} <>--{expression}
43                          -->{next}* & {MethodCall}-->{target}  t[3]
44           reportMap  t -> theElement(sendCall <>--{arguments}
45                                            <>--{next} -->{target}).name  end;
```

The query iterates over the Transition archetype tuples t. The variable container is bound to all blocks (StatementListContainer) contained in the method that contains the activation call of the next state one after the other, i.e., first it is bound to the method body, then to blocks of if, catch, or catch statements. The variable sendCall is in turn bound to all MethodCall vertices contained in the container. The first predicate in the with part of the query ensures that sendCall invokes in fact a Method with

name "send", and the second predicate ensures that in the block container the activation of the next state occurs. The query reports a map which assigns the name of the argument given to the send() method in sendCall to the selected Transition archetypes.

This was the last operation of the `ExtractStateMachines.gretl` transformation. The complete task could be solved with only 45 lines of transformation source code.

3 Conclusion

In this paper, the complete GReTL reference solution of the program understanding case has been discussed in details. In this conclusion, some statements about the evaluation criteria are made.

The solution covers the core as well as both extension tasks, so it is *complete*. The *correctness* of the solution has been validated with the three provided input models. Transforming them always results in the same target model, except for the (for this case irrelevant) order of elements in the models.

With respect to *conciseness*, only 45 lines of transformation source code for this non-trivial task is very good.

The *performance* of the solution is also good. On SHARE [4], all three input models can be transformed in less than two seconds.

With respect to *understandability*, one may argue that GReTL's conception of incrementally constructing the target schema and graph simultaneously, its traceability concept of archetypes and images, and the query language GReQL are not easy to grasp at first. However, they have proven being flexible and expressive, so it might be worth the initial steep learning curve.

References

[1] Jürgen Ebert & Daniel Bildhauer (2010): *Reverse Engineering Using Graph Queries*. In: *Graph Transformations and Model Driven Engineering*, LNCS 5765, Springer, pp. 335–362, doi:10.1007/978-3-642-17322-6_15.

[2] Jürgen Ebert, Volker Riediger & Andreas Winter (2008): *Graph Technology in Reverse Engineering, The TGraph Approach*. In R. Gimnich, U. Kaiser, J. Quante & A. Winter, editors: *10th Workshop Software Reengineering (WSR 2008)*, *GI Lecture Notes in Informatics* 126, GI, pp. 67–81.

[3] Florian Heidenreich, Jendrik Johannes, Mirko Seifert & Christian Wende (2010): *Closing the Gap between Modelling and Java*. In Mark van den Brand, Dragan Gasevic & Jeff Gray, editors: *Software Language Engineering*, *Lecture Notes in Computer Science* 5969, Springer, pp. 374–383, doi:10.1007/978-3-642-12107-4_25.

[4] Tassilo Horn: *SHARE demo related to the paper Solving the TTC 2011 Reengineering Case with GReTL*. http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi= Ubuntu_10.04_TTC11_gretl-cases.vdi.

[5] Tassilo Horn (2011): *Program Understanding: A Reengineering Case for the Transformation Tool Contest*. In Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors: *TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011*.

[6] Tassilo Horn & Jürgen Ebert (2011): *The GReTL Transformation Language*. In Jordi Cabot & Eelco Visser, editors: *Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, *Lecture Notes in Computer Science* 6707, Springer, pp. 183–197, doi:10.1007/978-3-642-21732-6_13.