1 Introduction

This paper discusses the GReTL solution of the TTC 2011 Hello World case. The submitted solution covers all tasks including the optional ones.

GReTL (Graph Repository Transformation Language, [HEar, HE11]) is the operational transformation language of the TGraph technological space [ERW08]. Models are represented as typed, directed, ordered, and attributed graphs. For interacting with the rest of the world, there are import/export tools for both EMF [SBPM09] as well as GXL.

GReTL’s operations are either called in plain Java using the GReTL API or in a simple domain-specific language. In this solution, only the DSL is used.

The elementary transformation operations follow the conception of incrementally constructing the target metamodel together with the target graph. When creating a new metamodel element, a usually set-based semantic expression is specified that describes the set of instances that have to be created in the target graph. This expression is described by a query on the source graph formulated using the GReQL querying language [EB10].

Of course, if the target metamodel already exists, there is no need to create it from scratch. So a set of further elementary transformation operations is provided that have no effect on the target metamodel but work only on the instance layer, i.e., the operations create new vertices and edges, and they set attribute values. In this challenge, the target metamodels are provided, and thus the solutions use only these instance level operations.

Besides these elementary (and usually out-place operations), GReTL offers a set of in-place operations that allow for deleting elements, merging vertices, or replacing a matched

[^http://www.gupro.de/GXL/]: http://www.gupro.de/GXL/
pattern with some subgraph similar to rules graph transformation languages like GrGen.NET [JBK10].

In the next section, all Hello Word case tasks are discussed, thereby explaining GReTL in some details.

2 Task Solutions

In this section, we’ll go through all tasks in sequence and explain the GReTL transformations and GReQL queries when they come along.

Running the solution. When logged into the SHARE image containing the GReTL solutions (Ubuntu_10.04_TTC11_gretl-cases), the solution of the hello world case is located in the directory `/Desktop/GReTL_TTC_2011_SOLUTIONS/HELLO_WORLD/`. The EMF import followed by the GReTL transformations and GReQL queries and finally the EMF export can be executed using the `run.sh` shell script.

All transformations and queries are located in the `transforms/` directory. The result graphs and their visualizations are generated into the `outputs/` directory, and finally they are converted back to EMF models in the `emf_out/` directory.

2.1 Hello World!

Task 1: Hello World. The first task is to create one single Greeting vertex. Its text attribute should be set to "Hello World". The transformation file is `1-hello-world.gretl`.

```grel
transformation HelloWorld1;

CreateVertices Greeting <= set(1);

SetAttributes Greeting.text <= map(1 -> 'Hello World');
```

Line 1 declares the name of the transformation.

In line 3, the `CreateVertices` operation is invoked. The argument specifies the type name of the vertices to be created, i.e., Greeting. After the arrow symbol, there is a GReQL query that is run on the source graph and should return an arbitrary set. For any member of that set, a new Greeting vertex will be created, and the mapping from set member to new vertex is saved and can be used in following operation calls.

Since this is a constant transformation, there is no source graph. The query simply evaluates to a set containing the integer number 1. Thus, one new Greeting vertex is created, and the mapping from the number 1 to that vertex is saved in a function corresponding to the target element type, i.e., `img_Greeting`. In this context, we call 1 the archetype of the new Greeting vertex, which is in turn the image of the number 1.
In line 4, the text attribute of Greeting vertices is set. The SetAttributes operation expects a map, which assigns to each archetype of some vertex or edge the value that its corresponding target graph image should have set for the specified attribute.

In this case, the map contains only one single entry: the integer 1 maps to the string ``Hello World''. Thus, for the image of the number 1 in img_Greeting, i.e., the new Greeting vertex, the text attribute is set to "Hello World".

**Task 2: Hello World Extended.** The second task is to create another hello world graph, but this time containing 3 vertices of different types and 2 edges connecting these vertices (2-hello-world.gretl).

```gretl
transformation HelloWorld2;

CreateSubgraph
  (GreetingMessage "$" | text = "Hello")
  <--{GreetingContainsGreetingMessage}
  (Greeting "$")
  -->{GreetingContainsPerson}
  (Person "$" | name = "TTC Participants")
<= set(1);
```

Here, the CreateSubgraph operation is used. Like CreateVertices it gets a GReQL query resulting in a set. For each member in that set, a subgraph specified by the template given as first parameter is created. In that template, several GReQL queries can be embedded, and the $ variable refers to the current member. Since the set contains only the number 1, the template will be evaluated only once, and thus the single binding of $ is 1.

The template graph specifies that there should be a GreetingMessage vertex with text = "Hello" that is connected to a Greeting vertex using a GreetingContainsGreetingMessage edge. That greeting should also be connected to a Person vertex with name = "TTC Participants" using a GreetingContainsPerson edge.

**Task 3: Model-2-Text.** In this task, a model-to-text transformation should be created, which prints the greeting of the graph created by Task 2. The canonical way to do that is to use a plain GReQL query (3-hello-to-text.greql).

```gretl
from greet: V[Greeting]
reportSet
  theElement(greet <--{GreetingContainsGreetingMessage}).text
  " "
  theElement(greet <--{GreetingContainsPerson}).name ++ "!"
end
```

For any Greeting vertex, a string is created by concatenating the text of that greeting’s GreetingMessage, one space, the name of that greeting’s Person, and finally an exclamation point. 
mark. The result is a set of strings. Since the graph contains only one greeting, it is a set with one single string ``Hello TTC Participants!''.

Such a query returns basic values, vertices, edges, sets, maps, lists, but no models. There’s an alternative GReTL solution that creates a graph conforming to the Result metamodel with one single StringResult vertex (3-hello-to-text.gretl).

1
transformation HelloWorld2StringResult;
2
// Create one StringResult per Greeting
3CreateVertices StringResult <= V{Greeting};
4
// Set the StringResult.result attributes
5SetAttributes StringResult.result <= from greet: keySet(img_StringResult)
6    reportMap greet ->
7        theElement(greet <=--{GreetingContainsGreetingMessage }).text
8          ++ " " ++
9        theElement(greet <=--{GreetingContainsPerson }).name ++ "!"
10end;

Basically, the same GReQL query is used for setting attributes, only that it returns a map to assign the value to the single StringResult vertex’s result attribute.

In the following, all querying task are implemented only as GReQL queries.

2.2 Count Matches with certain Properties

Task 4: Count Nodes. In this tasks, the number of Node vertices is to be determined (4-count-nodes.greql).

count(V{Node})

The expression V{VertexType} returns an ordered set of all vertices of that type, and the function count() returns the number of elements in a collection.

Task 5: Count Loops. The number of edges whose src and trg links point to the same Node should be counted.

Please note that because all models were imported from EMF without any optimization, in the TGraph, all elements of type Edge_ are in fact vertices, and the src and trg references are real edges of the types Edge_LinksToSrc and Edge_LinksToTrg. The edge type names were auto-generated, and Edge was renamed to Edge_, because in the default package the type names Graph, Vertex, and Edge are forbidden, because they would shadow the abstract default types of that name.

Anyway, the query is in the file 5-count-loops.greql.
First, all loops are determined by selecting those Edge_ vertices for which traversing the incident edge with src role leads to a vertex that equals the vertex reachable by traversing the incident edge with trg role. A tuple is returned that contains the number of loops and the set of loops.

**Task 6: Isolated Nodes.** To find the isolated nodes, the following GReQL query is used:

```grelq
let isolatedNodes := from n: V{Node}
    with degree{Edge_LinksToSrc, Edge_LinksToTrg}(n) = 0
reportSet n.name end
in "There are " ++ count(isolatedNodes) ++ " isolated nodes: " ++ isolatedNodes
```

First, all isolated nodes are determined by restricting the nodes to those, which are not connected to any Edge_LinksToSrc or Edge_LinksToTrg edge. Instead of the nodes themselves, the names are selected for a better comparison with the EMF models. As result a string is constructed that tells about the number of isolated nodes and lists them.

**Task 7: Circle of Three Nodes.** The following GReQL query is used:

```grelq
let circles := from n1, n2, n3: V{Node}
    with n1 <> n2 and n2 <> n3 and n3 <> n1
    and n1 <--> {Edge_} -->{trg} n2
    and n2 <--> {Edge_} -->{trg} n3
    and n3 <--> {Edge_} -->{trg} n1
reportSet n1.name, n2.name, n3.name end
in "There are " ++ count(circles) ++ " circles: " ++ circles
```

The variables n1, n2, and n3 iterate over all Node vertices. The with-part ensures they are pairwise distinct, and when starting at n1 an incoming edge has to traversed to some intermediate, anonymous Edge_ vertex, and from that, an outgoing edge leads to n2. The other two sides of the circle are specified likewise.

**Optional Task 8: Dangling Edges.** Here, “dangling edge” refers to a vertex of type Edge_, which has only one outgoing edge.

```grelq
let danglingEdges := from e: V{Edge_}
    with degree{Edge_LinksToSrc, Edge_LinksToTrg}(e) = 1
reportSet e end
```
The vertices of type Edge_ are restricted to those that have exactly one incident edge of type Edge_LinksToSrc or Edge_LinksToTrg.

### 2.3 Task 9: Reverse Edges

The task of reversing edges is done using a GReTL in-place transformation. Here, a new operation MatchReplace is used. It receives a template graph as first argument in which parentheses denote vertices and arrow symbols with curly braces denote edges. It also receives a GReQL query following the arrow symbol. The query results in a set, and for each member in the set (a match), the template graph is applied. The elements in the template graph may refer to things in the current current match via the variable $$. All elements in a match that are used in the template graph are preserved, all elements in a match that are not used are deleted.

```plaintext
transformation ReverseEdges;
MatchReplace (' theElement($[0] -->{src} ) ' ) <--{Edge_LinksToTrg}
  ('$[0]' ) -->{Edge_LinksToSrc} (' theElement($[0] -->{trg} ) ' )
<= from e: V[Edge_]
  reportSet e,
  edgesFrom{Edge_LinksToSrc, Edge_LinksToTrg}(e) end;
```

The query reports each Edge_ vertex and its outgoing source and target edges. For each match, the template graph is applied. In the template graph, $$[0]$$ references the Edge_ vertex in the current match tuple by its index. It is specified, that from that Edge_ vertex, a new Edge_LinksToTrg edge has to be created leading to the vertex that is the src at that time. Likewise, a new Edge_LinksToSrc edge has to be created to the vertex that is the trg at that time.

Because the set of Edge_LinksToSrc and Edge_LinksToTrg edges starting at the Edge_ vertex $$e$$ reported by the query is not used in the template graph, those old edges are deleted.

### 2.4 Simple Migration

**Task 10: Simple Migration.** This task is solved with an out-place GReTL transformation. In lines 3 to 5, the vertices of type Graph_, Node, and Edge_ are `"copied" over. More precisely, in line 3 we state, that for each Graph_ vertex in the source graph, a Graph_ vertex has to be created in the target graph. The same is done for the other two vertex types.

The CreateVertices operation automatically saves the traceability information, so that we can refer to that in following rules. For example, the mappings from source graph Node
vertices to target graph Node vertices is saved in a map img_Node (the *image* function), and
the reverse mappings are saved in a map arch_Node (the *archetype* function). Because the
target metamodel types Graph_, Node, and Edge_ are all subclasses of GraphComponent, the
maps img_GraphElement and arch_GraphElement contains all mappings of the individual
image and archetype functions.

```
transformation SimpleMigration;
CreateVertices Graph_ <= V(Graph_);
CreateVertices Node <= V(Node);
CreateVertices Edge_ <= V(Edge_);
SetAttributes GraphComponent.text <= from elem: keySet(img_GraphComponent)
  reportMap elem -> hasType{Node}(elem) ? elem.name : "" end;
CreateEdges Edge_LinksToSrc <= from e: E{Edge_LinksToSrc}
  reportSet e, startVertex(e), endVertex(e) end;
CreateEdges Edge_LinksToTrg <= from e: E{Edge_LinksToTrg}
  reportSet e, startVertex(e), endVertex(e) end;
CreateEdges Graph_ContainsGcs <= from e: E{Graph_ContainsNodes, Graph_ContainsEdges}
  reportSet e, startVertex(e), endVertex(e) end;
```

This is important when setting the GraphComponent.text attribute in lines 7 to 9. The
query returns a map that assigns to each GraphComponent archetype, i.e., a source graph
Graph_, Node, or Edge vertex, the value that its image in the target graph should have set
for the text attribute. If the archetype is of type Node, then the value is the contents of its
name attribute. Else, it is the empty string.

From line 11 on, the edges are "copied" into the target graph. For any source graph
Edge_LinksToSrc edge a target graph Edge_LinksToSrc edge is created. Because edges cannot
exist on their own, the query has to result in a set of triples. The first component is the
archetype of the new edge which can be used in following operations to refer to it. The sec-
ond and third component are the archetype of the new edge's start and end vertex. Thus,
the new target graph Edge_LinksToSrc edge starts at the image of its source counterpart's
start vertex and it ends at the image of the end vertex.

In lines 19 to 21, we create the Graph_ContainsGcs edges. Each of those edges corre-
sponds to either a source graph Graph_ContainsNodes or a Graph_ContainsEdges edge.

### Optional Task 11: Topology Chaining.

In this task, the conceptual edges represented as vertices should be transformed into real edges. The transformation is pretty similar to
the last one. The Graph_ and Node vertices but no Edge_ vertices are created in the target graph, the text attributes of nodes are set, and the Graph_ContainsNodes edges are created.

```plaintext
transformation TopologyChaining;
CreateVertices Graph_ <= V{Graph_};
CreateVertices Node <= V{Node};

SetAttributes Node.text <= from n: keySet(img_Node)
  reportMap n -> n.name end;

CreateEdges Graph_ContainsNodes <= from e: E{Graph_ContainsNodes}
  reportSet e, startVertex(e), endVertex(e) end;

CreateEdges NodeLinksToLinksTo <= from e: V{Edge_}
  reportSet e, theElement(e ->{src}),
  theElement(e ->{trg}) end;
```

The interesting operation is the last one, which creates the NodeLinksToLinksTo edges (the silly name is generated from the class and role names in the Ecore metamodel). For each source graph Edge_ vertex e, the query reports a triple containing e, the Node vertex at the src end of the outgoing Edge_LinksToSrc edge, and the Node vertex at the trg end of the Edge_LinksToTrg edge starting at e.

### 2.5 Delete Node with Specific Name

**Task 12: Delete Node n1.** In this task, all nodes with name attribute set to "n1" should be removed.

```plaintext
transformation DeleteNodeN1;
Delete <= from n: V{Node}
  with n.name = "n1"
  reportSet n end;
```

The Delete operation deletes all elements returned by the query.

**Task 13: Delete Node n1 and Connected Edges.** This task is similar to the previous task, except that all Edge_ vertices connected to the Node to be deleted should be deleted, too.

```plaintext
transformation DeleteNodeN1AndIncidentEdges;
Delete <= from n: V{Node}
  with n.name = "n1"
  reportSet n, -->{src, trg} n end;
```
So the query returns the nodes to be deleted and all Edge_vertices reachable by traversing edges targeting n where n is in the src or trg role.

### 2.6 Optional Task 14: Insert Transitive Edges

In this task, transitive edges should be created. We implemented this GReTL transformation as in-place transformation on graphs conforming to the “topology chaining” metamodel (Fig. 6 in the case description).

```
transformation TransitiveEdgesGraph3;

matches := from n1, n2, n3 : V{Node}
  with n1 <> n2 and n2 <> n3
  and n1 -->{linksTo} n2
  and n2 -->{linksTo} n3
  and not n1 -->{linksTo} n3
reportSet n1, n3 end;

Iteratively MatchReplace ('$[0]' -->{NodeToToLinksTo} ('$[1]' )
  <= from n : matches
  with not n[0] -->{linksTo} n[1]
reportSet n[0], n[1] end;;
```

This transformation uses a little trick. We only want to create transitive edges, but not transitive edges of transitive edges and so forth until we get a graph where every node is connected with any other node. Thus, we compute the matches beforehand.

The query binds three nodes to the variables n1, n2, and n3 and ensures that they are distinct. This is not totally clear by the task description, but if they were not distinct, then transitive edges could be loops, which doesn’t seem too sensible. The query reports a set of pairs, and for each pair, a new NodeToToLinksTo edge between n1 and n3 should be created.

Iteratively is a higher-order operation that executes the following transformation operations as long as any of them is applicable.

The query of the MatchReplace call iterates over the pairs of nodes and checks if no transitive edge created by a previous iteration exists. In that case, the template graph specifies the creation of such an edge.

The little trick is required for this reason: Although the query provided to MatchReplace results in a set of matches, the operation skips matches containing elements that already occurred in previous matches. These elements might have been modified and might be invalid. So we need more than one iteration here, but since the transitivity should be applied to only the initial state of the graph, we have to compute matches before.
References


