Saying Hello World with GReTL – A Solution to the TTC 2011 Instructive Case

Dipl.-Inform. Tassilo Horn

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

This paper discusses the GReTL solution of the TTC 2011 Hello World case [7]. The submitted solution covers all tasks including the optional ones.

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. There are import/export tools for EMF models and metamodels. GReTL is designed as a Java API, but a simle concrete syntax is provided as well.

The elementary GReTL transformation operations follow the conception of incrementally constructing the target metamodel together with the target graph. In this challenge's tasks, the target metamodels are provided, so only operations working on the instance level are used, i.e., the operations create new vertices and edges in the target graph, and they set attribute values. GReQL queries [3] evaluated on the source graph are used to specify what has to be created.

Besides these elementary out-place operations, GReTL offers a set of in-place operations that allow for deleting elements or replacing elements matched by a pattern with some subgraph similar to rules of graph replacement systems. In the next section, all mandatory Hello Word tasks are discussed, thereby explaining GReTL in some details. The solutions of the optional tasks are discussed in the appendix.

2 Task Solutions

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

Task 1: Hello World. The first task is to create one single Greeting vertex.

```
1 transformation HelloWorld1;
2 CreateVertices Greeting <== set(1);
3 SetAttributes Greeting.text <== map(1 -> 'Hello World');
```

Line 1 declares the name of the transformation. In line 2, the CreateVertices operation is invoked. The argument specifies the type of the vertices to be created, i.e., Greeting. After the arrow symbol, there is a GReQL query that is evaluated on the source graph and should return an arbitrary set. For any

Van Gorp, Mazanek and Rose (Eds.):
Fifth Transformation Tool Contest (TTC 2011)
EPTCS 74, 2011, pp. 295–301, doi:10.4204/EPTCS.74.24

member of that set, a new Greeting vertex is 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 evaluates to a set containing only the number 1. Thus, one new Greeting vertex is created. The mapping from 1 to that vertex is saved in a function corresponding to the target element type (img_Greeting). In this context, 1 is called the *archetype* of the new Greeting vertex. The new Greeting vertex is in turn called the *image* of 1.

In line 3, 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 an extended hello world graph.

```
1 transformation HelloWorld2;
2 CreateSubgraph (GreetingMessage '$' | text='"Hello"')
3                  <--{GreetingContainsGreetingMessage}
4                (Greeting '$')
5                  -->{GreetingContainsPerson}
6                (Person '$' | name = '"TTC Participants"') <== set(1);
```

The CreateSubgraph operation is similar to CreateVertices in that it gets a GReQL query resulting in a set. For each member in that set, a subgraph specified by the template preceeding the <== symbol is created. Parenthesized constructs denote vertices to be created, and arrow symbols with curly braces denote edges to be created. In each template vertex or edge, first the type name is given, then an archetype, followed by an optional list of attribute-value pairs. For edges, the archetypes may be omitted, but for vertices the archetype is mandatory, because it is used internally to refer to the new vertices when creating the edges connecting them. The $ variable refers to the current member of the set returned by the query. Since the set contains only one member, the template will be evaluated only once, and the single binding of $ is 1. Three vertices (a GreetingMessage, a Greeting, and a Person) and two connecting edges (a GreetingContainsGreetingMessage edge, and a GreetingContainsPerson edge) are created. The archetypes of all vertices is the value of $, i.e., 1.

Task 3: Model-2-Text. A GReQL query is used to serialize the graph created in the last task.

```
1 from greet: V{Greeting}
2 reportSet theElement(greet <>--{GreetingContainsGreetingMessage}).text
3           ++ " " ++
4           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 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!". The expression greet <>--{GreetingContainsPerson} calculates the set of vertices reachable from greet by traversing a containment edge of type GreetingContainsPerson in the direction from part to whole. Because there must be exactly one person associated to a greeting by such an edge, the function theElement() is used to extract it.

Task 4: Count Nodes. In this task, the number of Node vertices is to be determined.

```
count(V{Node})
```

Task 5: Count Looping Edges. Note that because all models were imported from EMF without any optimization, in the TGraph all elements of type Edge_[1] are in fact vertices, and the src and trg references are real edges of the types Edge_LinksToSrc and Edge_LinksToTrg.

```
let loops := from e: V{Edge_} with e -->{src} = e -->{trg} reportSet e end
in tup(count(loops), loops)
```

First, all loops are determined by selecting those Edge_ vertices whose src and trg edges point to the same vertex. A tuple is returned that contains the number of loops and the set of loops. This is not required by the task, but it should be noted that the result tuple could be processed in Java similar to a SQL result set.

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

```
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 mentions the number of isolated nodes and lists them.

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

```
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 form a circle. For example, between n1 and n2 there has to be a vertex of type Edge_ which references n1 and n2. The query results in a string mentioning the number of circles and lists them.

Task 9: Reverse Edges. The task of reversing edges is done using a GReTL in-place transformation. Here, the operation MatchReplace is used. Just like the CreateSubgraph operation used in Task 2, it receives a template graph. Analogously, it receives a GReQL query following the <== 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

[1] The underscore has been appended because Edge is the abstract base type of all edge types and thus a reserved word.

not used are deleted, and elements of the template graph that don't reference an element in the match are created.

```
1 transformation ReverseEdges;
2 MatchReplace ('$[1]') <--{Edge_LinksToTrg} ('$[0]') -->{Edge_LinksToSrc} ('$[2]')
3 <== from e: V{Edge_}
4     reportSet e, endVertex(srcEdge), endVertex(trgEdge), srcEdge, trgEdge end
5     where srcEdge := theElement(edgesFrom{Edge_LinksToSrc}(e)),
6           trgEdge := theElement(edgesFrom{Edge_LinksToTrg}(e));
```

The query reports a set of 5-tuples, one tuple for each Edge_ vertex, including its source and target nodes, and the corresponding Edge_LinksToSrc and Edge_LinksToTrg edge.

For each match, the template graph is applied with the current match tuple bound to $. $[0] references the match's Edge_ vertex by its index in the reported match tuple using an array-like syntax. Likewise, $[1] references the source node, and $[2] references the target node. Since these nodes are referenced, they are preserved. The edges in each match tuple ($[3] and $[4]) are not referenced in the template and thus deleted. Two new Edge_LinksToTrg and Edge_LinksToSrc edges are created, but this time the former source node is the target and the former target node is the source.

Task 10: Simple Migration. This task is solved with an out-place GReTL transformation. In lines 2 to 4, the vertices of type Graph_, Node, and Edge_ are "copied" over, i.e., for any Graph_ node in the source graph, a Graph_ node in the target graph is created and likewise for Node and Edge_ nodes.

```
1  transformation SimpleMigration;
2  CreateVertices Graph_ <== V{Graph_};
3  CreateVertices Node <== V{Node};
4  CreateVertices Edge_ <== V{Edge_};
5  SetAttributes GraphComponent.text
6   <== from elem: keySet(img_GraphComponent)
7       reportMap elem -> hasType{Node}(elem) ? elem.name : "" end;
8  CreateEdges Edge_LinksToSrc
9   <== from e: E{Edge_LinksToSrc} reportSet e, startVertex(e), endVertex(e) end;
10 CreateEdges Edge_LinksToTrg
11  <== from e: E{Edge_LinksToTrg} reportSet e, startVertex(e), endVertex(e) end;
12 CreateEdges Graph_ContainsGcs
13  <== from e: E{Graph_ContainsNodes, Graph_ContainsEdges}
14      reportSet e, startVertex(e), endVertex(e) end;
```

Then the GraphComponent.text attribute is set in lines 5 to 7. 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 8 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 given to CreateEdges 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 second 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 12 to 14, the Graph_ContainsGcs edges are created. Each of those corresponds to either a source graph Graph_ContainsNodes or a Graph_ConstainsEdges edge.

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

```
1 transformation DeleteNodeN1;
2 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.

```
1 transformation DeleteNodeN1AndIncidentEdges;
2 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.

3 Conclusion

In this paper, the solutions of all Hello World tasks have been briefly discussed. Most transformations were implemented using the elementary GReTL operations CreateVertices, CreateEdges, and SetAttributes whose concepts are explained in more details in [6]. GReTL is a very extensible language, and the in-place operations MatchReplace, Iteratively, and Delete used in some task solutions were added to the language for solving the *Compiler Optimization* case [1, 5].

References

[1] Sebastian Buchwald & Edgar Jakumeit (2011): *Compiler Optimization: A Case for the Transformation Tool Contest*. In Van Gorp et al. [8].

[2] J. Ebert, V. Riediger & A. 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] 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.

[4] Tassilo Horn: *SHARE demo related to the paper Saying Hello World with GReTL – A Solution to the TTC 2011 Instructive Case*. `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_gretl-cases.vdi`.

[5] Tassilo Horn (2011): *Solving the TTC 2011 Compiler Optimization Case with GReTL*. In Van Gorp et al. [8].

[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.

[7] Steffen Mazanek (2011): *Hello World! An Instructive Case for the Transformation Tool Contest*. In Van Gorp et al. [8].

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

A Appendix: Optional Tasks

In this short appendix, the three optional tasks are discussed.

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

```
let danglingEdges := from e: V{Edge_}
                     with degree{Edge_LinksToSrc, Edge_LinksToTrg}(e) = 1
                     reportSet e end
in "There are " ++ count(danglingEdges) ++ " dangling edges: " ++ danglingEdges
```

The vertices of type Edge_ are restricted to those that have exactly one incident edge of type Edge_LinksToSrc or Edge_LinksToTrg. Those are counted and listed.

Optional Task 11: Topology Changing. In this task, the conceptual edges represented as vertices should be transformed into real edges. The transformation is quite 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.

```
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 strange name is generated by our Ecore importer 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. Thus there will be a new NodeLinksToLinksTo edge for every Edge_ vertex which starts/ends at the Node vertices the original Edge_ referenced.

Optional Task 14: Insert Transitive Edges. In this task, transitive edges should be created. It is implemented as in-place transformation on graphs conforming to the "topology changing" metamodel (Fig. 6 in the case description [7]).

```
transformation TransitiveEdgesGraph3;
matches := from n1, n2, n3: V{Node}
           with n1 <> n2 and n2 <> n3 and n3 <> n1
             and n1 -->{linksTo} n2
             and n2 -->{linksTo} n3
             and not n1 -->{linksTo} n3
           reportSet n1, n3 end;
Iteratively MatchReplace ('$[0]') -->{NodeLinksToLinksTo} ('$[1]')
  <== from n: matches with not n[0] -->{linksTo} n[1] reportSet n[0], n[1] end;;
```

This transformation uses a little trick in order to create only transitive edges of the original graph, but not the transitive closure. Therefore, the set of matches is calculated beforehand. This is a set of Node pairs where a transitive edge has to be created in between.

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 occured in previous matches. These elements might have been modified in a way that invalidates the current match.