Generating Sierpinski Triangles by the Tiger EMF Transformation Framework

Gabriele Taentzer[1] and Enrico Biermann[2]

[1] Philipps-Universität Marburg, Germany
taentzer@mathematik.uni-marburg.de
[2] Technische Universität Berlin, Germany
enrico@cs.tu-berlin.de

1 Introduction

This paper describes how Sierpinski triangles can be generated on basis of the Tiger EMF Transformation Framework (EMT). See [3] for a definition of Sierpinski triangles and for a description how to construct them. In our solution, Sierpinski triangles are considered as Eclipse Modeling Framework (EMF) models [1]. EMT is an Eclipse plug-in which adapts graph transformation concepts to realize EMF model transformations. Thus, EMT is based the Eclipse plug-in EMF. We define an EMF model and EMT transformation rules to generate Sierpinski triangles, and control their generation process by a Java program. For an installation guide and a user manual for EMT we refer to [2].

2 The Meta model

The meta model for Sierpinski triangles is given by the EMF model in Fig. 1. It just consists of one object type which is *Vertex*. A speciality of EMF models is the containment relation. All objects belonging to the model have to be contained (transitively) in the root object.

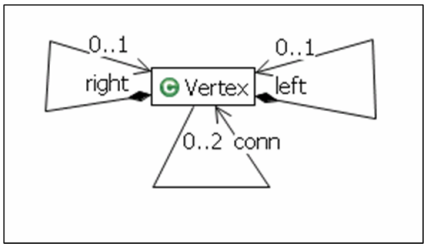

Fig. 1. The Sierpinski meta model as EMF model

Hence, we consider Sierpinski triangles as hierarchical object structures. Each object has two containment relations, i.e. *left* and *right* for the left and right

children. Additionally, a normal association *conn* is defined which is used to represent mainly the horizontal object relations in Sierpinski triangles. Since each object belongs to at most one container, some objects do not have left children, but two objects linked by *conn*.

After having defined an EMF model for the Sierpinski case, Java code for creating, modifying, storing, and loading model instances can be created. Moreover, a complete tree-based editor can be generated which we will use to create the start model for the generation process. The resulting Sierpinski triangle will be again an EMF instance model which can also be considered by this editor, although the tree-like presentation is not really insightful.

3 The Model Transformation Rules

To generate Sierpinski triangles two EMT rules are needed. We can use the visual rule editor of EMT to define them. The first rule depicted in Fig. 2 is used, if root object of the corresponding sub-triangle has a left child. Otherwise, the second one in Fig. 3 is used. Please note the negative application condition of this rule which prohibits the root having a left child.

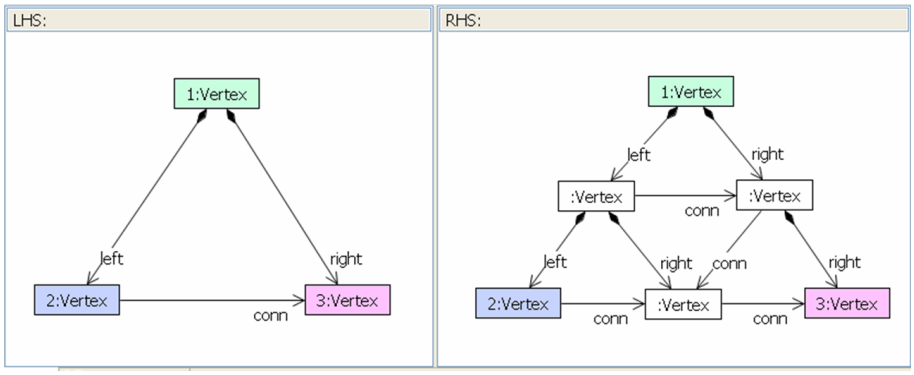

Fig. 2. Rule *AddTriangles* for generating Sierpinski triangles - if root object has a left child

After having defined both rules, a rule project can be generated which contains all Java classes needed to perform EMF model transformations. In our case, two classes *AddTrianglesRule.java* and *AddTriangles2Rule.java* are generated (among others).

4 The Application control

The EMF model for starting the generation process looks like the left-hand side of rule *AddTriangles* in Fig.2. To reach the first refinement we have to apply rule

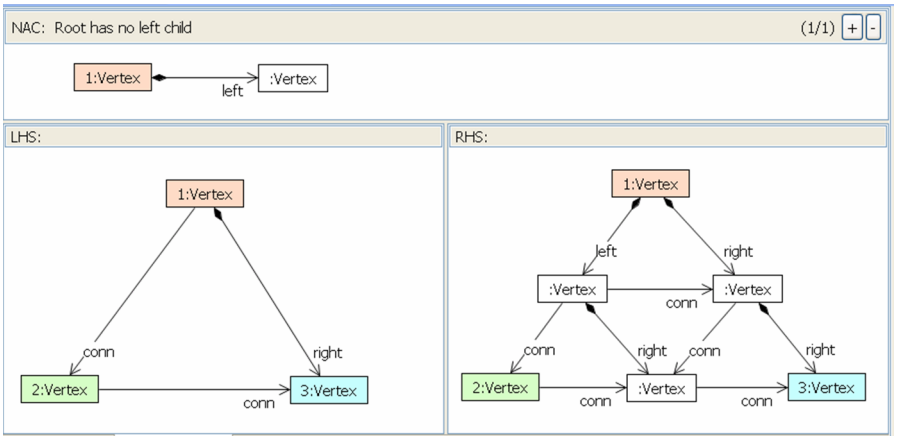

Fig. 3. Rule *AddTriangles2* for generating Sierpinski triangles - if root object has no left child

AddTriangles just once. The result graph would look like the right-hand side of this rule. The next refinement is already more complex. Here, the three outer triangles are refined, the two left ones by rule *AddTriangles*, while the right one is refined by rule *AddTriangles2*. These three applications have to be performed, before the next refinement level is considered. Therefore, we need an application control which acts level-wise. On each level, we perform the generation process as follows: Since containment relations *left* and *right* define a spanning tree, we use these relations to navigate through the model. Starting at the root vertex, its triangle is refined first. Thereafter, the refinement process continues at its left child, if existing, and at its right one as long as existing. The refinement process terminates for a certain level, if the bottom objects which do not have any children, are reached.

Consider the following code fragment for controlled refinement by EMF model transformations. For each rule application, first a new rule instance has to be created. Then, a partial match can be set which is done by e.g. method *setVertex0* which maps the upper vertex in the left-hand side in Figs. 2 and 3, resp. Thereafter, the rule is executed.

```
void applyTriangles1(SierpinskiImpl s, Vertex root, Vertex currentVertex){
     AddTrianglesRule addtr = new AddTrianglesRule(currentVertex);
     addtr.setVertex0(currentVertex);
     Vertex leftVertex = currentVertex.getLeft();
     Vertex rightVertex = currentVertex.getRight();
     addtr.execute();
     if (leftVertex.getLeft() != null)
         s.applyTriangles1(s,root,leftVertex);
     else if (leftVertex.getConn().size() > 1)
          s.applyTriangles2(s, root, leftVertex);
     if (rightVertex.getRight() != null)
          s.applyTriangles2(s, root, rightVertex);
   }

void applyTriangles2(SierpinskiImpl s, Vertex root, Vertex currentVertex){
     AddTriangles2Rule addtr2 = new AddTriangles2Rule(root);
     addtr2.setVertex0(currentVertex);
     Vertex leftVertex = null;
     if (currentVertex.getConn().size() > 0) {
         leftVertex = (Vertex)currentVertex.getConn().get(0);
         Vertex rightVertex = currentVertex.getRight();
         addtr2.execute();
         if (leftVertex.getLeft() != null)
             s.applyTriangles1(s,root,leftVertex);
         if (rightVertex.getRight() != null)
             if (rightVertex.getLeft() == null)
                 s.applyTriangles2(s,root,rightVertex);
             else s.applyTriangles1(s, root, rightVertex);
     }
}

public static void main(String[] args) {
    // Initialization to be added
    for(int i=1; i<= LEVEL; i++){
        System.out.println("Level: "+i);
        startTime = System.currentTimeMillis();
        s.applyTriangles1(s,root,root);
        elapsedTime = System.currentTimeMillis() - startTime;
        System.out.println("Elapsed time: " + elapsedTime);
    //file output to be added
    }
```

5 Timing results

In the following table we provide the time usage in milliseconds for the generation process of Sierpinski triangles. These figures have been obtained on a PC with an Intel(R) Core(TM)2 2.33GHz processor and 2GB of main memory. Although not explicitly counted in the program shown above, this table also shows the number of nodes generated at each level.

Generation level	Time usage (in ms)	# nodes
1	16	6
2	0	15
3	0	42
4	16	123
5	62	366
6	250	1095
7	2000	3282
8	16500	9843
9	227250	29526
10	3604984	88575

Table 1. Time usage and number of nodes

6 Conclusion

Although the generation process for Sierpinski triangles we have chosen, is rule-based, non-determinism has been eliminated nearly completely by the external application control. For each rule application, the match of vertex 1 is explicitly determined. Having matched vertex 1, the match of the left-hand side is completely determined in most cases. This applies to both rules in Sec. 3 and leads to a quite fast solution. Comparing this solution with other ones where the application control is put mainly into the rules, it is much faster. No wonder, pure rule-based solutions usually have much higher matching costs, since they consider the whole graph as matching domain in each transformation step.

References

1. Eclipse Modeling Framework. http://www.eclipse.org/emf, 2007.
2. Tiger EMF Model Transformation Framework. http://cs.tu-berlin.de/emftrans, 2007.
3. R. Geiß, C. Mallon, and M. Kroll. Sierpinski Triangle for the AGTIVE 2007 Tool Contest. http://www.informatik.uni-marburg.de/∼ swt/agtive-contest/, 2007.