Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools

Gabriele Taentzer[1], Enrico Biermann[2], Dénes Bisztray[3], Bernd Bohnet[4], Iovka Boneva[5], Artur Boronat[3], Leif Geiger[6], Rubino Geiß[7], Ákos Horvath[8], Ole Kniemeyer[9], Tom Mens[10], Benjamin Ness[11], Detlef Plump[12], and Tamás Vajk[8]

[1] Philipps-Universität Marburg, Germany, taentzer@mathematik.uni-marburg.de
[2] Technische Universität Berlin, Germany, enrico@cs.tu-berlin.de
[3] University of Leicester, UK, {dab24,aboronat}@mcs.le.ac.uk
[4] Universität Stuttgart, Germany, bohnet@informatik.uni-stuttgart.de
[5] University of Twente, The Netherlands, bonevai@cs.utwente.nl
[6] Kassel University, Germany, leif.geiger@uni-kassel.de
[7] Universität Karlsruhe, Germany, rubino@ipd.info.uni-karlsruhe.de
[8] Budapest University of Technology and Economics, Hungary, ahorvath@mit.bme.hu,tamas.vajk@aut.bme.hu
[9] BTU Cottbus, Germany, okn@informatik.tu-cottbus.de
[10] University of Mons-Hainaut, Belgium, tom.mens@umh.ac.be
[11] Vanderbilt University, US, bness@isis.vanderbilt.edu
[12] The University of York, UK, det@cs.york.ac.uk

Abstract. In this paper, we consider a large variety of solutions for the generation of Sierpinski triangles, one of the case studies for the AGTIVE graph transformation tool contest [15]. A Sierpinski triangle shows a well-known fractal structure. This case study is mostly a performance benchmark, involving the construction of all triangles up to a certain number of iterations. Both time and space performance are involved. The transformation rules themselves are quite simple.

1 Introduction

The field of graph transformation was set up over 30 years ago, but the development of supporting tools started with considerable delay. Currently, a number of tool environments for different graph transformation approaches is available and the activity in tool development has increased considerably. Thus, a comparison of tools with respect to both functional and non-functional issues is becoming more and more important.

Graph transformation tools can serve very different purposes. The case study we consider in this paper allows us to compare the efficiency of graph representations and the performance of repeated rule applications. For this comparison we have chosen the generation of Sierpinski triangles. Due to its exponential nature, the problem involves graphs which are getting huge within a few generation steps. Theses graphs need not be typed and attributed; hence very simple graph models may be used. Furthermore, the generation process is very regular and can be performed with only a few rules.

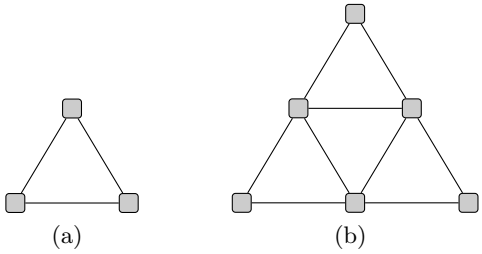


Fig. 1. Initial and first generation of the Sierpinski triangle

In the context of the AGTIVE tool contest, the response to the call for this case study has been impressive. Twelve solutions with variants have been submitted, differing heavily in the underlying graph transformation approaches and tools, the graph representation, and the application control for rules. At the end of this paper, we categorize the given solutions and compare their runtime performance.

This paper is structured as follows: The case study used for competition is presented in Section 2. It comprises the generation of Sierpinski triangles. Section 3 gives an overview on the dimensions of solutions, while Section 4 presents a variety of concrete solutions. In Section 5, we briefly compare the presented solutions and draw some conclusions.

2 Case Study "Generation of Sierpinski Triangles"

The goal of this case study is to measure the performance of graph transformation tools constructing Sierpinski triangles. The Sierpinski triangle is a fractal named after Waclaw Sierpinski who described it in 1915. Originally constructed as a mathematical curve, this is one of the basic examples of self-similar sets, i.e. it is a mathematically generated pattern that can be reproduced at any magnification or reduction.

An algorithm for obtaining arbitrarily close approximations to the Sierpinski triangle is as follows:

1. Start with an equilateral triangle with a base parallel to the horizontal axis.
2. Shrink the triangle by $\frac{1}{2}$, make two copies, and position the three shrunk triangles so that each triangle touches each of the two other triangles at a corner.
3. Repeat step 2 with each of the smaller triangles.

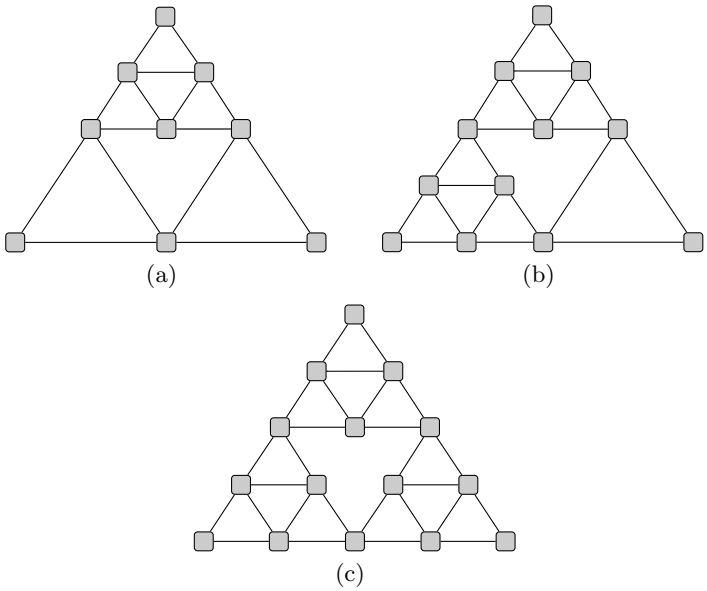


Fig. 2. Second generation of the Sierpinski triangle

2.1 Sierpinski Triangle as a Graph

For the purpose of the case study the Sierpinski triangle has to be represented as a (mathematical[13]) graph, and the construction rules have to be restated in the context of graph transformation. As an initial step a triangle has to represented as a graph: Three nodes connected by three edges (see Figure 1(a)).

Next, an elementary "Sierpinski step" is defined: On every edge of a triangle a node is placed. These new nodes are connected by edges (see Figure 1(b) in comparison to 1(a)). This forms a triangle consisting of four smaller triangles. The inner of the four triangles is considered "dead" and no further "Sierpinski steps" will be performed there. The other three triangles are candidates for further steps. (See Fig. 2 for the next elementary Sierpinski steps.) If all elementary steps are done for a certain graph (without reconsidering newly created triangles), we call this a generation (Figure 1(a) shows generation zero, Figure 1(b) shows generation one, and Figure 2(c) shows generation two). It is required for this case study that a generation is completed before any transformation for the next generation takes place.

2.2 Goals of the Case

This case study is pretty easy to implement: It uses only small pattern graphs, simple graph rewrites, and only a few rules. The generated graphs get huge fast.

[13] A mathematical graph has no immediate representation on a two dimensional plane—though embeddings may be computed.

The number of nodes is equal to $\frac{3}{2}(1+3^n)$ and the number of edges is $3^{(n+1)}$ with n being the number of generations. So it tests the ability of a tool to represent large graphs efficiently, and to perform simple rewrites, fast. With growing number of generations it is possible to sample memory usage and computation time. Last but not least, we can see how the tools are capable of enabling adequate meta models, rule sets, and rule applications.

3 Overview on Solutions

The solutions presented in the following differ heavily. For getting a better overview, we discuss those dimensions of modelling that play a role for this case study.

3.1 Modelling Choices

The solutions presented below differ heavily concerning the representation of graphs and modelling of Sierpinski steps. In the following, we list the main alternatives.

Graph representation In most solutions, graph nodes and edges are typed and carry information important for the generation process, i.e. they mostly guide the generation process. Some solutions also use additional nodes or edges to store data about intermediate steps of the Sierpinski triangle generation. Moreover, also node attributes are used for storing additional information. Clearly, additional graph elements and attributes may affect the efficiency of graph representation.

Modelling of Sierpinski steps All solutions contain one or more rules for performing an elementary Sierpinski step. All generation steps, except the first one, consist of several applications of the elementary step. The solutions show different kinds of controlling rule applications. Basically, we can distinguish parallel from sequential rule application.

The generation of Sierpinski triangles is well suited for parallel rule application and means that the basic Sierpinski step is performed on all triangles being "alive" simultaneously (compare Sec. 2.1).

Sequential rule application necessarily leads to intermediate graphs where some atomic triangles of the current generation step are already refined, while others still have to be considered. For not refining an already refined triangle again in the same generation step, some application control has to be added which can be done in different ways. Some solutions add further graph elements holding information about the generation process and/or add application conditions to their generation rules or even add further rules, while others rely on external control which is formulated by e.g. regular expressions. Besides just controlling the selection of rules, some solutions even control the rule matching explicitly and thus eliminate any kind of non-determinism in rule application.

3.2 Graph Transformation Approaches

A number of different graph transformation approaches are used to perform the generation of Sierpinski triangles. The solutions proposed differ also according to offered graph transformation features. In the following, we sketch the most important features for this case study:

- Nearly all solutions use typed graphs, sometimes even with node type inheritance.
- Some solutions are based on attributed graphs. In these cases simple attribute computations are performed only.
- The approaches differ in the ability to visually or textually represent graph rules.
- While most approaches offer sequential rule application only, there are some approaches which support parallel rule application (in addition). Here, a rule is applied to all possible matches in parallel. Note, that parallel refers to simultaneous rewriting semantics; it has not necessarily to be implemented by parallel threads of any kind. In fact almost all tools do not support thread-based or distributed rewriting.
- To further control sequential rule application, additional application conditions for rules are offered by several approaches. These include negative application conditions, attribute conditions, and type conditions.
- Another form of application control is to put control on top of rules by using concepts from regular expressions, abstract state machines, activity diagrams, Java programs, and recursive rule application. Some of these forms allow to control rule matches in addition.

4 Solutions

In the following, twelve different graph transformation solutions for the generation of Sierpinski triangles are presented. These solution are available with all details at the following newly created Web site for graph transformation cases: http://gtcases.cs.utwente.nl/.

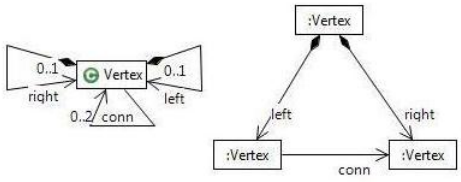
4.1 Tiger EMF Transformation Framework

The Tiger EMF Transformation Framework (EMT) [16] is a tool for modeling and applying graph transformation rules. The solutions consist of a set of graph transformation rules on EMF [8] models, that are designed using the Visual Editor of EMT. The production rules are defined by rule graphs, namely a left-hand side (LHS) and a right-hand side (RHS). The rule set is compiled to Java code and run by the Eclipse development platform [7]. This enables the implementation of control structure to perform the specified changes to the given model instance.

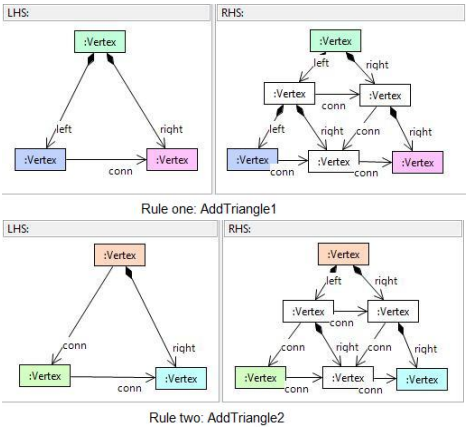
Two solution were implemented, a deterministic and a non-deterministic one.

Deterministic solution This solution uses a programmed control flow to apply rules.

For our solution we first define an EMF model and an initial Sierpinski triangle as shown on the right. This can be defined by using any editor for EMF models, for example the EMF tree editor.



Afterwards we define two rules for transforming a Sierpinski instance such that we reach a new Sierpinski generation. This can be done by using the graphical rule editor which is part of EMT. The defined rules are translated to Java code that changes a given Sierpinski triangle as described by the rules *AddTriangle1* and *AddTriangle2*. In our case we need two rules because EMF requires a containment hierarchy between all classes and a class can only be contained in exactly one other class. Rule *AddTriangle1* refines left triangles where the upper vertex contains both lower vertices, while *AddTriangle2* refines right triangles. Here, the upper vertex contains the lower right vertex only.



In the next step we define a control structure for the application of our two rules. AddTriangle1 should be applied to the uppermost vertex and if possible to all left children. AddTriangle2 should be applied to all right children if possible. Both rules are no longer applicable when attempting to apply AddTriangle1 or AddTriangle2 to the vertices at the bottom of the containment hierarchy. After the control structure terminates, the resulting model instance is a new Sierpinski generation.

Non-Deterministic Approach The non-deterministic solution uses the algorithm from [12] to generate the Sierpinski triangles.

The metamodel shown in Figure 3 was used. There are two types of nodes. The *NormalNode* is an ordinary node in the Sierpinski structure, while the *CentralNode* is used to mark those triangles that needs unfolding in the next Sierpinski step. The *Alive* property of *CentralNode* indicates if that triangle is alive in that step.

The Sierpinski step is implemented with the rule depicted in Figure 4. A triangle that has an *alive CentralNode* gets unfolded. The generated triangles

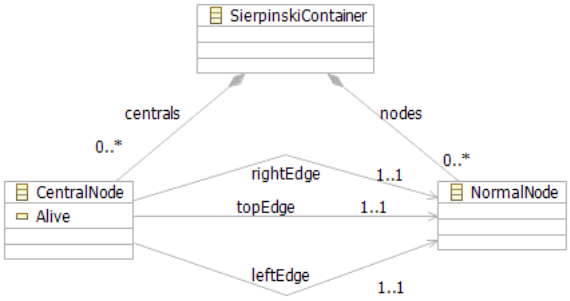


Fig. 3. Metamodel

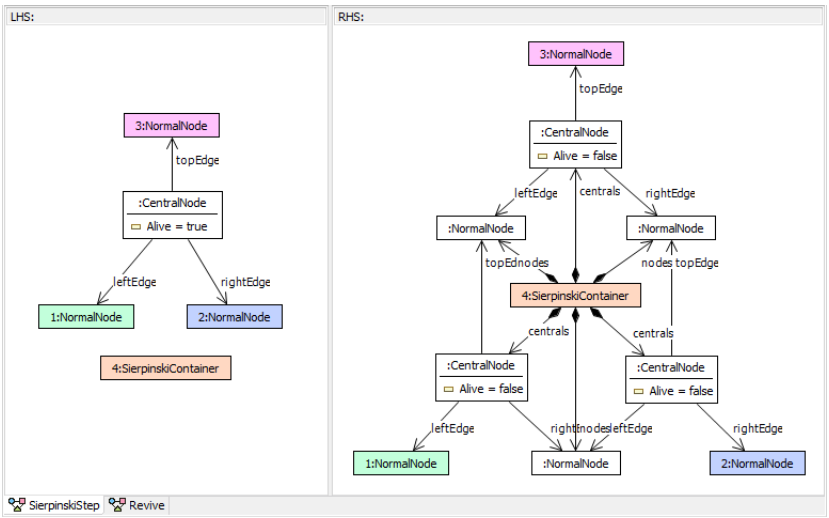


Fig. 4. Sierpinski Step

are not *alive*. When all alive triangles have been unfolded, a second *aliveRule* revives them for the next step. These two phases form the Sierpinski step.

4.2 Graph Transformation using Two Tapes

Using two tapes in graph transformation has many advantages since the rule interpreter can apply rules in parallel. During the rule application, the input graphs stay unchanged while the rule interpreter builds up new graphs. Therefore, rules can access the input graphs as well as the graphs created so far. In the case that rules need the result of other rules they form a sequence. Later applied rules might apply in parallel with other rules in the same state. The context (conditions) determines the application order of these rules and not an explicit specification of the order.

We originally developed the approach for Natural Language Processing (NLP) where we use it for instance to generate texts by mapping text plans to semantic graphs, semantic graphs to syntactic graphs, and syntactic graphs to topological graphs, cf. [5]. The demands for the graph transformation language come from the area of NLP where the rule interpreter can do much in parallel because of the nature of the application, but not everything. An example for this is the mapping of syntactic graphs to topological graphs that we apply to determine the word order. The words for example of all noun phrases (such as 'the blue car') can be ordered independently of other noun phrases and therefore in parallel whereas distinct complete phrases of German sentences are ordered depending on the position of the main verb. These rules determine first the position of the verb and depending on the result, the position of the other parts. Within the same graph transformation approach, we could easily describe the mapping of the Sierpinski triangle with one rule. The rule interpreter applies rules in four steps. (1) First, it searches with a parallel matching algorithm all occurrences of the left-hand side in the input graph and evaluates the conditions. (2) Then the rule interpreter clusters the rules, which are applicable together. In the case of the Sierpinski triangle, the rule interpreter builds only one cluster since no alternative rules are specified. (3) After that, the rule interpreter creates the right-hand sides of the rules for each matched rule in parallel. (4) Finally, the rule interpreter glues the graph fragments together.

The approach works very well and fast for the problem of the Sierpinski triangle. It builds the thirteenth generation with one core of a CPU in 18.9 seconds and the speed nearly doubles to 10.3 seconds with four cores of the same CPU. Looking at the figures, the question comes up, why does the processing speed not increase even more? Partially essential is that the gluing step is not performed in parallel and compared with the gluing the parallel matching works in this application on a three times smaller data set, therefore it contributes much less to the processing time. Another part of the answer seems related with the used CPU itself due to a test, which identified the memory bandwidth as a problem. The cause could be that the CPU is a first generation quad core, which is blamed as a not 'native' quad core. Nevertheless, the parallel approach is very promising since in future CPUs with many more cores will become standard and many applications such as the evolution of plants or Natural Language Processing fits very well to parallel approaches, which are capable to compute, like our brain, solutions in parallel.

4.3 The Groove Tool

Groove is a graph transformations tool-set based on the SPO approach for untyped, edge-labelled, simple graphs. The tool-set comes with a GUI allowing to easily define transformation rules and graphs, and to apply graph transformation either interactively, or automatically using so called exploration strategies. Strategies can also be used without the graphical interface. Although Groove is a general purpose graph transformation tool, it is optimised for generating (a finite portion of) all possible derivations in a graph grammar and allows to verify

properties on the set of derived graphs and on the derivation paths. Groove is Java-based, so platform independent. The tool, as well as the solution presented here, can be downloaded at [1].

Let us now explain the main characteristics of our modelling of the Sierpinski triangles in the Groove framework.

Modelling the triangles and the rules. Fig. 5 represents an intermediary graph met while computing the second generation.[14] The bottom part represents the Sierpinsky fractal itself encoded by a set of triangles in a very straightforward way. The top part is additional control structure present in the graph. It is composed by generation nodes numbered from 0 to 4 (for computing the fourth generation), and a "current" marker indicating the currently computed generation. Each elementary triangle "belongs" to the generation on which it was constructed, encoded as an additional edge from its top node to the corresponding control node. On Fig. 5, the two big elementary triangles belong to the first generation, and the three small elementary triangles belong to the current, second generation. This is used to ensure that a particular generation is completed before the computation of the next one starts. Transformation is ensured by two small and simple rules. The first one performs an elementary step: a triangle belonging to the previous generation is replaced by three new triangles of the current iteration. On Fig. 5, such elementary step was just performed on the bottom right triangle. The second rule is with lower priority, thus applicable only if the first one does not match. It simply moves the "current" marker to the control node corresponding to the next generation. None of the rules uses negative application conditions.

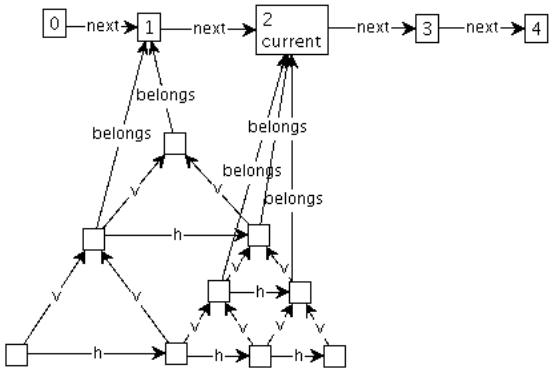


Fig. 5. Intermediary graph while computing the second iteration.

Computing the fractal. Transformations in Groove are fully non-deterministic, on purpose. However, in the case of the Sierpinski triangles, rule applications are confluent. We used the existing *Linear exploration strategy* to ensure

[14] Node labels on that figure are just another representation of self-edges.

that only a single derivation path is computed, thus avoiding superfluous rule transformations. Moreover, the tool allows to specify in which order edge labels should be matched, thus guided finding of matches. In the case of the Sierpinski triangles this allowed to avoid the computation of any superfluous matches.

Performance. Memory is a critical resource in Groove. As mentioned previously, the tool computes the set of all graphs derivable in a graph grammar and stores all intermediate results. Different memory-saving mechanisms are in place, as for instance sharing nodes and edges between graphs. They showed quite efficient, as we managed to compute the twelfth iteration of the Sierpinski fractal on a desktop machine with 1,5 GB of memory and in 45 seconds.

Alternative solution using quantified transformations. Since its last version, Groove implements so called quantified transformation rules. Roughly speaking, and among other things, quantified rules allow to make atomic the application of a rule for all of its matches in the host graph. In the case of the Sierpinski triangles, one can define a single, quite compact rule that is responsible for the computation of one generation. This results in slightly better performance, but the main improvement is the increased expressiveness of transformation rules. More information on quantified transformations can be found in the Groove documentation accessible at [1].

4.4 MOMENT2-GT

MOMENT2-GT is a graph transformation tool based on the SPO approach. Graphs are provided as EMF-based models so that their nodes are attributed and typed, taking inheritance into account. Graph transformation definitions are constituted by a set of production rules, which are defined in a QVT-based textual format, where OCL expressions can be used either as guards in (possibly negative) application conditions or as attribute value manipulation expressions. In MOMENT2-GT, a graph transformation definition is compiled into a rewrite theory in Maude [6]. MOMENT2-GT permits defining production rules as deterministic (production equations) or non-deterministic (production rules). The inclusion of this explicit difference allows performing model checking of graph-based systems where states are algebraically defined by means of a metamodel and production equations, and transitions between such states are defined as production rules. In EMF, bidirectional and containment edges can be defined. MOMENT2-GT takes into account such features to avoid the generation of inconsistent EMF models with dangling edges. This consistency checking can be disabled if dangling edges are explicitly avoided in the transformation definition. The tool and the solution presented here are available at [3].

In this solution, we have based ourselves on the transformation that is provided in [12], also implemented with the Tiger EMF Transformer (see the non-deterministic solution in Section 4.1). Our transformation definition consists in two simple rules: *a)* a first rule computes the division of a triangle and *b)* a second rule ensures that the following iteration in the fractal generation process is not performed until all triangles have been split. The second rule contains a negative application condition.

In MOMENT2-GT, the input graph is represented as a term of a specific sort that is defined in a rewrite theory, and the execution of a graph transformation is handled by Maude's algorithm for term rewriting modulo associativity and commutativity. Maude finishes the graph rewriting process when it achieves a normal form. The resulting term is parsed by MOMENT2-GT and projected as an EMF model again.

Although our tool is based on the reuse of Maude's term matching algorithm without taking into account optimized rewriting strategies, we have shown that MOMENT2-GT can be used to rewrite reasonably big graphs but efficiency needs to be improved still. The advantage of our approach relies on the reuse of Maude-based formal verification techniques [6] for graph transformations together with modeling standards.

4.5 Fujaba Solution

This section reports on our case study with the Fujaba environment, cf. [2], on building Sierpinsky triangles. It turned out, that the key bottleneck for building Sierpinski triangles is the memory usage. Thus, we exploited Fujaba's code generation features for the implementation of unidirectional to-one associations. Our model uses objects for the vertices of triangles with only three unidirectional to-one associations to refer to the horizontal right neighbor and to the vertical left and right neighbor. This models the triangles, appropriately, with a small memory footprint. Actually, we found out that two outgoing edges per node are sufficient. Thus, our model can even be reduced by one edge. But the rules for this solution are a bit more complex, so we won't explain this solution here due to space constraints.

We have decided to use a recursive approach for building the triangles. Figure 6 shows that recursive rule. The rule has to be executed on the top node of a triangle. This node is represented by the `this` node in our rule. From that node the `left` edge is traversed to get the left node of the triangle and the `right` edge to get the right node. To represent the triangle structure the `horizontal` edge is checked as well. Note, this could be omitted for further performance tuning. Now a new triangle is created into the one found before. This is indicated by the elements marked with ≪create≫. Since we use to-one links creating a new link automatically destroys the old link. So we left out the ≪destroy≫ markers for performance reasons. When the current triangle has been refined, we initiate the refinement of the left and the right underlying triangles. This is specified using a recursive call on the `left` and the `right` object. Note that, if the lookup for the triangle in this rule is not successful, the modifying operations are not executed. That results in termination of the recursion if the bottom triangles have been refined.

To initialize the algorithm an initial triangle has to be created. On the top node of the triangle the `refine` method described above can be called several times depending on the number of iterations one wants to calculate. This execution is modeled by a second rule not shown here. From these rules Fujaba now generates standard Java code which can be used for performance measuring.

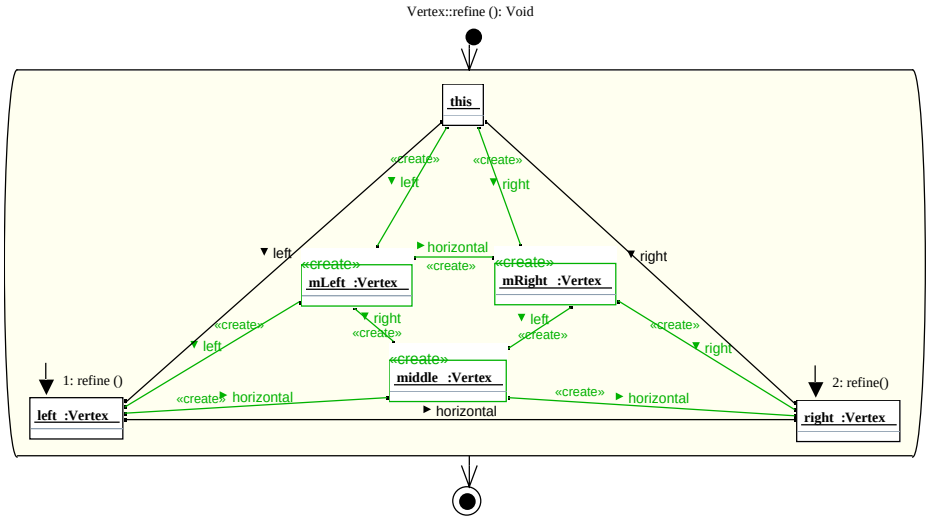


Fig. 6. Refining Rule for Sierpinski triangles

Measurement data We have done our measurements on a 64-bit quad core AMD processor with 32GB memory using Java6. 15 iterations take about 1.4 seconds. Each additional iteration takes about three times longer until the memory is full. On our machine this works up to iteration 17 (12 seconds). We found that in our model the runtime consumption for graph pattern lookup is neglectable since we use a recursive approach that passes the important handle elements to the next rule application. Thus runtime is dominated by creating new objects and assigning pointer values. Since our approach seemed fast enough for us, we tried to optimize the memory footprint to be able to do some more iterations. Fujaba uses objects to represent nodes and fields of those objects to represent edges. So only the objects need memory. On a 32-bit machine, each of our nodes has three pointers which take 4 bytes each. A Java object has an additional pointer to its class and a unique ID which are both again 4 bytes each. So one node costs 20 bytes. On our 64-bit system the memory usage doubles. To get rid of the 16 bytes of Java internal memory usage, we wrote a C++ code generation for Fujaba. C++ objects do not have the unique ID and if all methods are declared final also no pointer to their class. That way we were able to reduce the memory usage to 12 bytes per node. So, using C++ one more iteration fits into our memory. But the generated C++ code was a bit slower than the Java code. 17 iterations took 22 seconds and 18 took 1:08 minute. Using the algorithm mentioned above which needs only two pointers per node, we were even able to reduce the memory footprint to 8 bytes per node for the C++ code. The rules needed here just need one additional distinction of cases which does not make them measurably slower.

4.6 GrGen.NET

In contrast to semi-automatic tools like Fujaba, that require the user to specify the starting points of pattern matching, GrGen.NET—using the search plan driven approach to graph pattern matching [4]—can compute this information on its own. Hence the solution for GrGen.NET (version 1.3.1) does not involve predefined matches, although the tool supports rules with parameters (for more information on the tool see [10]).

The meta model:

```
node class A;
node class B;
node class C;

node class AB extends A,B;
node class BC extends B,C;
node class CA extends C,A;

edge class E0;
edge class E1;
```

The rules:

```
rule init {
 pattern {
 }
 replace {
  a:A -:E0-> b:B -:E0-> c:C -:E0-> a;
 }
}
```

```
rule gen0 {
 pattern {
  a:A -:E0-> b:B -:E0-> c:C -:E0-> a;
 }
 replace {
  a -:E1-> ab:AB -:E1-> b -:E1-> bc:BC
    -:E1-> c -:E1-> ca:CA -:E1-> a;
  ab -:E1-> ca -:E1-> bc -:E1-> ab;
 }
}
```

```
rule gen1 {
 pattern {
  a:A -:E1-> b:B -:E1-> c:C -:E1-> a;
 }
 replace {
  a -:E0-> ab:AB -:E0-> b -:E0-> bc:BC
    -:E0-> c -:E0-> ca:CA -:E0-> a;
  ab -:E0-> ca -:E0-> bc -:E0-> ab;
 }
}
```

Fig. 7. The meta model and all rules for the GrGen.NET-based solution

As a first step we generate an initial triangle with the rule `init` (see Figure 7). Afterwards we use two almost identical rules (`gen0, gen1`) in an alternating fashion. Each rule is applied as long as possible, thus computing a whole generation. The graph rewrite sequence `init & (gen0* & gen1*)[5]` produces the 10th generation (see [10]).

The edge types `E0` and `E1` are used for distinction of generations, i.e. the rule `gen0` replaces `E0` edges with `E1` edges and vice versa. The node types together with the orientation of the edges ensure that only appropriate (not dead) triangles are matched; in particular node types encode the positions. We use multiple inheritance to cope with the situation, where one node occurs in two triangles in different positions.

We also experimented with parallel rewriting semantics. This way only one rule is needed to refine the initial triangle, because the interlocking of steps is done automatically. Analogous to the graph rewrite sequence above, we get `init & [gen][10]`. Due to the overhead of temporarily storing all matches, this approach takes 20% more time.

Because GrGen.NET always supports all features like multi-edges, complex type hierarchies as well as attributed nodes and edges, performance-wise suboptimal code is produced. Note that the information present in a meta model (like *connection assertions*) is rich enough to automatically generate a stripped down and therefore more efficient implementation (see LIMIT in Section 5.2).

4.7 Viatra2

The current section highlights the concepts used to implement the Sierpinski triangles example in the Viatra2 [9] (VIsual Automated model TRAnsformations) framework. Viatra2 is a general-purpose model transformation engineering framework that aims at supporting the entire life-cycle, i.e. the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains in the MDA.

Metamodel and Transformation. From the programmers point of view, the most difficult part of implementing the Sierpinski triangle generator is to create the correct triangle *"finder mechanisms"*. In our solution, we tried to adhere to the typing scheme found in the problem description by taking advantage of the multiple-inheritance support of the Viatra2 framework resulting the metamodel depicted in Fig. 8.

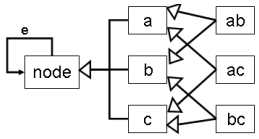
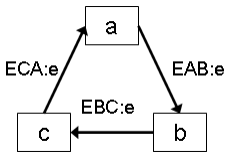


Fig. 8. The metamodel of the Sierpinski triangles

This representation allowed us to create a very simple and elegant pattern – used in Viatra2 to describe the *precondition* (LHS and the NACS) of a GT rule – illustrated in Fig. 9, where the right and left side describe the Viatra Textual Command Language (VTCL) representation and the graphical notation, respectively. Capital letters stand for variables, normal letters denote direct references to modelspace elements. For instance the expression `a(A)` declares that the variable `A` is of type `a`. On the other hand, the expression `node.e(ECA,C,A)` means that the variable `ECA` refers to an edge of type `node.e` that points from the entity in `C` to `A`. The pattern in Fig. 9 matches a triangle with vertices type `a,b,c`, respectively. The order is granted by the direction of the arrows.




```
pattern triangle(A,B,C,EAB,EBC,ECA) =
 //Nodes
 a(A); b(B); c(C);
 //Edges
 node.e(EAB,A,B); node.e(EBC,B,C);
 node.e(ECA,C,A);
```

Fig. 9. Sierpinski pattern

As for the model manipulation part, instead of using *GT rules* for the triangle generations we were utilizing *graph patterns* to match the corresponding model parts and then performing the model manipulation by built in ASM rules. This resulted in overall better memory consumption and a slightly faster runtime performance.

Conclusion. Note that, we also used the example for profiling the upcoming VIATRA2 release and it pointed out that the most critical part in the pattern matching process is the *model manager* as it took more than 99 percent of the total execution time. As for the overall performance, we were happy to see that our current *interpretation* based engine can handle up to 800000 model elements (level 11) within reasonable time.

4.8 Solution Using XL

The XL solution [11] is very simple as rules are applied in parallel by default, which exactly matches the Sierpinski construction. The complete rule is:

```
a:LLVertex -e0-> b:Vertex -e120-> c:Vertex -e240-> a ==>>
    a -e0-> ab:LLVertex -e0-> b -e120-> bc:Vertex -e120-> c
        -e240-> ca:LLVertex -e240-> a,
    ca -e0-> bc -e240-> ab -e120-> ca;
```

The class `Vertex` represents vertices, its subclass `LLVertex` those vertices which are the lower left vertex of a black triangle in the usual 2D representation. Furthermore, we use edge types `e0`, `e120`, `e240` where the number is the angle of the edge in the 2D representation. Using `LLVertex` speeds up matching since a match for the pattern exists if and only if `a` is a lower left vertex of a black triangle. Thus, we exclude dead ends of pattern matching as soon as possible.

We tested the performance of our solution in four different settings. The simplest one uses the graph model of the modelling environment GroIMP [11]. This introduces some amount of memory overhead as nodes store additional bookkeeping information and all changes to the graph are logged in a protocol. On a 3 GHz computer with 2 GB RAM we were able to execute 13 steps, the last step took 17.3 seconds on average. But logging can be deactivated which is our second setting and allowed an additional step. This setting is also used for the benchmark in Figure 15. A significant improvement concerning memory

consumption and speed (roughly factor 3 in both respects) is achieved by the third setting where we use our own minimal graph model whose nodes only store three pointers to adjacent nodes, thus indirectly representing the edges (which can no longer be traversed bidirectionally). On 32-bit Java virtual machines, such a node requires 20 bytes. We were able to execute 15 steps, the last step took 54.7 seconds on average. The fourth setting reduces the memory consumption to 8.25 bytes (66 bits) for lower-left vertices and 0 bytes for the other ones on both 32- and 64-bit machines: at first, we dispense with the `e120` edge so that only a subgraph of the Sierpinski graph is generated (but which can be extended to the true graph by local operations). Furthermore, we address vertices by unique `long` values which in case of lower-left vertices are indices into a list of pairs of 33-bit values which hold the addresses of the neighbours. 33 bits are sufficient for up to 20 steps, the final graph then consumes nearly 27 B of memory. On a 2.6 GHz computer with 32 GB RAM (thanks to Andreas Hotho, University of Kassel) we were able to execute 20 steps where the last step took 2 hours on average, resulting in a graph with 5,230,176,603 nodes and 6,973,568,802 edges.

Being able to use XL for any given graph structure, even exotic ones as in the last setting, is certainly a highlight. We have to admit that the last setting is very tricky and requires temporary proxy objects in order to be accessible for XL.

4.9 AGG

The graph transformation tool *AGG* was developed at TU Berlin to explore advanced graph transformation properties for doing formal analysis. Performance was not one of its main design criteria. *AGG* can be used in two different modes: (a) via its built-in GUI to specify and execute graph transformations visually; (b) via its API to write Java programs that use AGG's underlying graph transformation engine. We explored both ways to implement the Sierpinski triangle.

GUI with rule sequences and NACs. Our first implementation resorts to *AGG*'s ability to define *rule sequences*, i.e., a predefined composition of graph transformation rules. We used the following rule sequence:

```
( FindMatch{*} ApplyToMatch{*} ){n}
```

where integer value `n` represents the desired number of iterations, and {`*`} denotes that each rule is applied as long as possible. The two transformation rules in this rule sequence are shown in Figure 10. `FindMatch` identifies and annotates all triangles that need to be expanded. It uses a negative application condition to avoid applying the rule more than once to the same occurrence. `ApplyMatch` performs the actual transformation on the found matches. This solution has two shortcomings. First, it requires an auxiliary, somehow artificial, node `X`. Second, the timing results are not very promising.

GUI with parallel matching. To improve performance, Olga Runge extended the *AGG* engine with a mechanism of *parallel matching*, enabling the computation of all possible matches of a given rule at once, and then repeatedly applying the desired transformation to all of these matches. For the Sierpinski example, this feature allowed us to simplify the solution by using only one rule (Figure 11), while simultaneously improving the timing results.

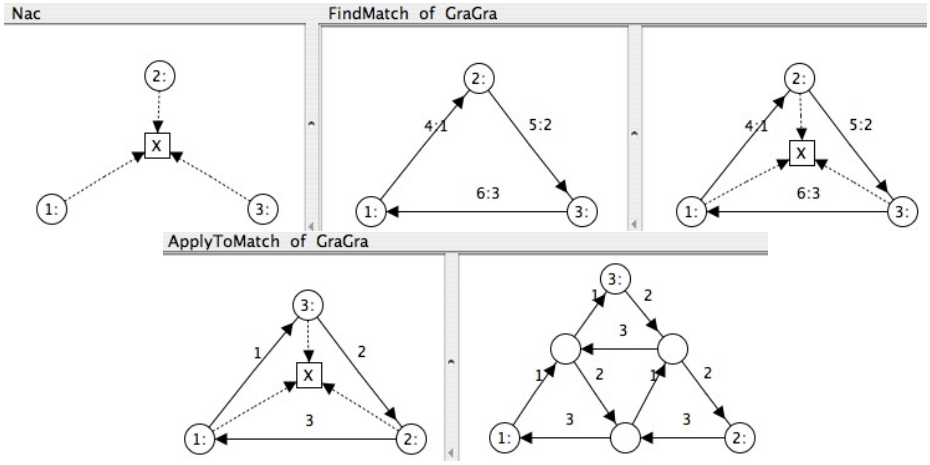


Fig. 10. Implementation of Sierpinski in AGG with NACs and rule sequences.

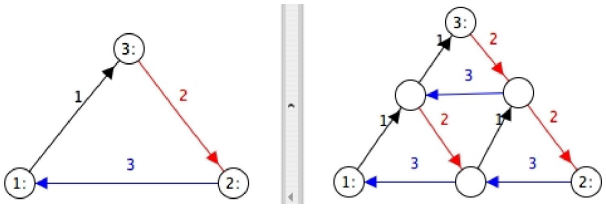


Fig. 11. Implementation of Sierpinski in AGG using parallel matching.

API-based solution. For our third and final implementation, we wrote a simple Java program that calls AGG's API to execute the graph transformations. This solution also relied on a single rule (Figure 11) using parallel matching, but was more complex to implement since one needs to write a Java program and know how the API works.

Timing results. We compared performance of the above three solutions.We observed that the GUI-based solution relying on parallel matching, as well as the API-based equivalent are the most performant. Still, even if the results are visualised on a logarithmic time scale to account for the fact that Sierpinksi generation is an inherently exponential problem, the curves remain exponential, whereas we would have expected a (theoretically) linear increase instead. It is therefore clear that the performance of AGG can still be increased considerably.

4.10 GReAT Solution

One solution was developed using GReAT, a meta-model based model transformation tool. GReAT, along with the Generic Modeling Environment, GME, are well-suited for this application because GME allows one to quickly implement domain-specific languages, and GReAT supports the quick implementation of model transformations in the form of graph transformations.

The overall process we used was the following. We started by defining a meta model in GME to describe models of Sierpinski Triangles. Next, we created an instance model of this meta-model that contained one triangle. We then created a GReAT transformation that performed the Sierpinski Triangle-generation algorithm on this instance model using simple graph rewriting rules that are explicitly sequenced.

The meta model consisted of a base object, SierpinskiTriangleModel. This object serves as the root container of the model as well as the container for all other objects. The "Generations" attribute on this object indicates the number of times the Sierpinski algorithm is to be performed on the input graph. The "DecrementGeneration" attribute is used during the transformation to determine when the "Generations" attribute is ready to be decremented. "Node" objects are used to represent triangle vertices, and "Connection" objects are used to represent the edges between triangles.

Using GReAT, a simple graph transformation was written to generate the Sierpinski triangle graph. The transformation takes an input graph containing a single triangle and produces the corresponding Sierpinski generation as a separate output file.

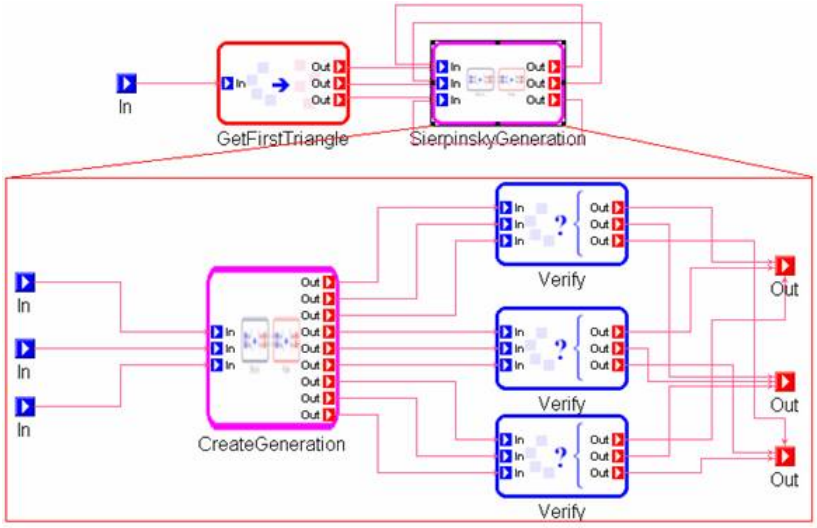


Fig. 12. GReAT rules

The transformation consists of several simple rules that are organized into hierarchical blocks. The first rule, "GetFirstTriangle" (see Figure 12), locates the triangle in the input model. The next block, "SierpinkiGeneration", is a sequence of rules that, when executed, will rewrite the graph into the next Sierpinski generation. "DecrementRule" (a rule contained inside the "CreateGeneration"

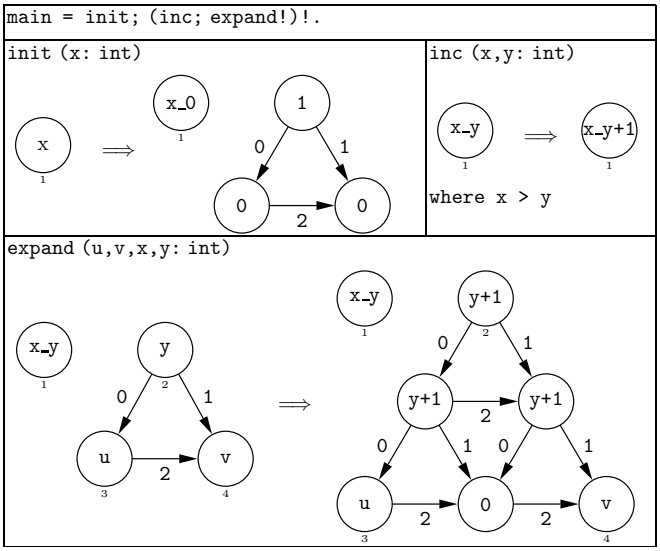


Fig. 13. GP program

block) is executed once per generation rather than once per triangle. After every execution of the "CreateGeneration", the number of generations remaining is compared to zero in the Verify test case. Once the "Generations" attribute equals zero, the Verify test case will fail, and the transformation will end.

Using GME and GReAT to perform this transformation provided a very simple, graphical approach that was very easy to implement and execute while still achieving an acceptable level of performance. In addition, smaller generations are very easy to visualize using the GME interface.

4.11 Generating Sierpinski Triangles with GP

The graph programming language GP is based on conditional rule schemata [14]. The program in Figure 13 consists of three rule-schema declarations and the main command sequence following the key word `main`. It expects as input a graph consisting of a single node labelled with the generation number of the Sierpinski triangle to be produced.

The rule schema `init` creates the Sierpinski triangle of generation 0 and turns the input node into a unique "control node" whose label is of the form `x_y`. The underscore operator allows to hold the required generation number `x` and the current generation number `y` in a common node.

After `init` has been applied, the nested loop `(inc; expand!)!` is executed. Intuitively, the operator `!` executes a subprogram as long as possible. In each iteration of the outer loop, the rule schema `inc` increases the current generation number if it is smaller than the required number. The latter is checked by the condition `where x > y`. If the test is successful, the inner loop `expand!` performs

a Sierpinski step on each triangle whose top node is labelled with the current generation number: the triangle is replaced by four triangles such that the top nodes of the three outer triangles are labelled with the next higher generation number. The test `x > y` fails when the required generation number has been reached. In this case the application of `inc` fails and hence the outer loop terminates and returns the current graph. The resulting graph is the Sierpinski triangle of the required generation.

The GP compiler translates the program of Figure 13 into bytecode that can be executed by the York abstract machine [14]. The execution times shown in Figure 15 have been obtained on a PC with an Intel Pentium 4 processor with a clock rate of 2.80GHz and 512MB of main memory. For these executions, the backtracking mechanism of the abstract machine has been switched off because the program's input/output behaviour is deterministic.

Most of the execution time is used to create the elements of the triangles while matching the left-hand sides of the rule schemata is fast due to the uniqueness of the control node labelled with `x_y`. In general, matching starts at the rarest node or edge and then proceeds to find other elements of the left-hand side of the rule schema. In the case of `expand`, clearly the rarest element is the control node. Next the variable `y` is bound, and so the root of the triangle labelled with `y` is found. Matching then extends over the 1- or 2-edge outgoing from the `y`-node, and finds the remainder of the structure in a unique way.

4.12 VMTS Solution

In Visual Modeling and Transformation System (VMTS) [18], we have created a metamodel that defines two types; the *Vertex* type is used to represent the nodes of the Sierpinski triangle, while the *DepthLimit* node helps to set the actual magnification (the depth level used in the transformation). The edges of the triangles are represented at meta level by a loop edge connected to the *Vertex*. Both types have an attribute *Level*. In case of a *Vertex* the attribute shows on which level (in which generation) the vertex has been created, while in case of the *DepthLimit*, *Level* means the depth that we would like to reach during the transformation.

The main idea behind our transformation is that at every step, we match all possible triangles in the complete Sierpinski triangle and we transform each triangle into four sub-triangles. There is only one exception: we do not process triangles in which the *Level* attributes of the forming nodes are equal. This exception is required to skip the inner triangles. There are two important comments on the procedure: (i) We do not match triangles for which the vertices are not direct neighbors; (ii) on the first (0^{th}) level, when we have only one triangle, the *Level* attributes of the nodes are the same. We can handle this as an initial step and avoid applying the exception rule in this case. The transformation stops if there is a node in the triangle for which the *Level* attribute is equal to the level set by the *DepthLimit* object. The input model of the transformation is an initial triangle having three vertices and a *DepthLimit* object.

In VMTS, we use an activity diagram-like specification [13] to define the steps of graph transformations. Our Visual Control Flow Language supports processes (transformation rules), start/end states, decisions, fork and join constructs. The utilized control flow, shown in Fig. 14, forms a pretest loop, which is executed until we reach the configured depth limit. The loop consists of two rewriting rules

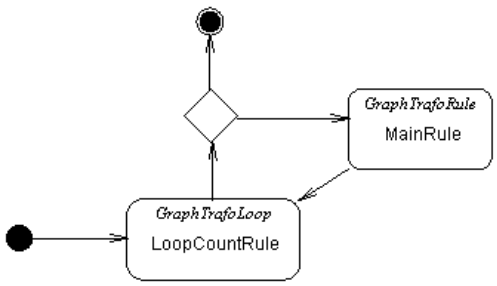


Fig. 14. Control flow diagram of the transformation in VMTS

and a decision node. The first rule, named *LoopCountRule* in the control flow model, references the *GraphTrafoLoop* transformation which checks whether the *Level* attribute of *DepthLimit* is 0. If so, the rule deletes the object, which makes the decision element select the end state of the transformation. Inside the loop, the main transformation rule is executed (*MainRule*, *GraphTrafoRule*). Matching is based on graphical rule definition, while rewriting uses Imperative OCL [17] code. The main rule is set to apply matching in a *MultipleMatch* mode, thus, the matching algorithm finds not only the first match, but all possible matches. This means that rewriting is applied once on the complete list of matches, which can dramatically improve performance according to our experiences. Moreover, this way the rule is executed only once for each iteration, which makes handling the *Level* attribute much simpler.

Our initial solution matched each triangle three times with different orientation and the filtering was applied after the matching process. With an improved matching which checks for multiplicity, we have achieved processing times of 23220ms on the 9^{th} level and 211750ms on the 10^{th}. It is important to note that our transformation framework applies the transformation steps in a strict order, defined by the control flow, and it does not compile control flows, but interprets them. Therefore, the processing times achieved can be considered as raw values without any optimization specific to this task.

5 Lessons Learned

For a rough comparison of the solutions presented, we give some key characteristics in the following subsection. They are meant to provide a basis for the subsequent performance comparison. Although concrete numbers are set into

relations, this comparison is supposed to give just a rough idea of runtime performances. The reader has to keep in mind that these performance tests have been executed on different computers and within different settings.

5.1 Approaches and Features

Even for a rough performance comparison of graph transformation tools it is important to know the degree of non-determinism used within the given solution. We distinguish the following kinds of rule matching and application to characterize the given graph transformation solutions:

1. Selection of rules and matches by the tool
2. The order of rule application is (partly) given, but matches are selected by the tool.
3. The order of rule application as well as their matches are (partly) given.
4. The order of rule application is (partly) given and rules are allowed to be matched and applied as often as possible in parallel.

In addition, we provide some information on the kind of graph representation used within the presented solutions. Some tools allow a custom graph representation to perfectly adapt to the case. In certain cases, graph features such as attributes, are not needed. Custom graph representations allow to adapt to such special cases.

To get a rough idea of the graph size in memory, we collected figures for the size of one elementary triangle, i.e. the size of three nodes glued together to a triangle produced in an elementary step of the Sierpinski generation.

A further information which seems to be significant for performance comparisons of graph transformations is the representation of edges. Here we look for double linked edges such that it is possible to traverse the edges in O(1) in both directions by just knowing an adjacent node. (Note: This feature has nothing to do with directed or undirected edges in the meta model.)

5.2 Runtime Performance

Comparing the performance of the solutions we can see a widespread distribution from milliseconds to hours for the same task—even spreading over several complexity classes (see Figure 15). To investigate how fast the fastest solutions really are, Mallon and Geiß developed a hand-coded solution called LIMIT. This artificial solution is roughly 2.6 times faster and 2.5 times more memory efficient than the fastest semi-automatic tool (Fujaba)(kind 3). The fastest fully automatic tools, i.e. GrGen.NET (kind 2) and parallel tools (XL, Two Tapes)(kind 4), are even two orders of magnitude slower than LIMIT. One of the differences between semi and fully automatic tools is that the semi-automatic ones require the developer to specify the starting points of pattern matching, whereas automatic tools can compute this information on their own. However, LIMIT uses only knowledge automatically deducible from a meta model. Hence it should be possible to tune tools to generate such efficient code automatically.

tool	kind	custom graph	size of triangle	edges doubly linked
TIGER EMF	1/3	no		
MOMENT2-GT	1	no		no
GROOVE	1	no		yes
TWO TAPES	1	partially	240 bytes	yes
FUJABA	3	yes	60 bytes	no
GRGEN.NET	2	in v2.0	312 bytes	yes
VIATRA	2	yes		yes
XL (GroIMP graph)	1	possible	336 bytes	yes
AGG	1/4	no		yes
GREAT	2			
GP	2	no	410 bytes	yes
VMTS	2	no	3396 bytes	yes
LIMIT	3	yes	6 pointers	no

Table 1. Some special solution characteristics.

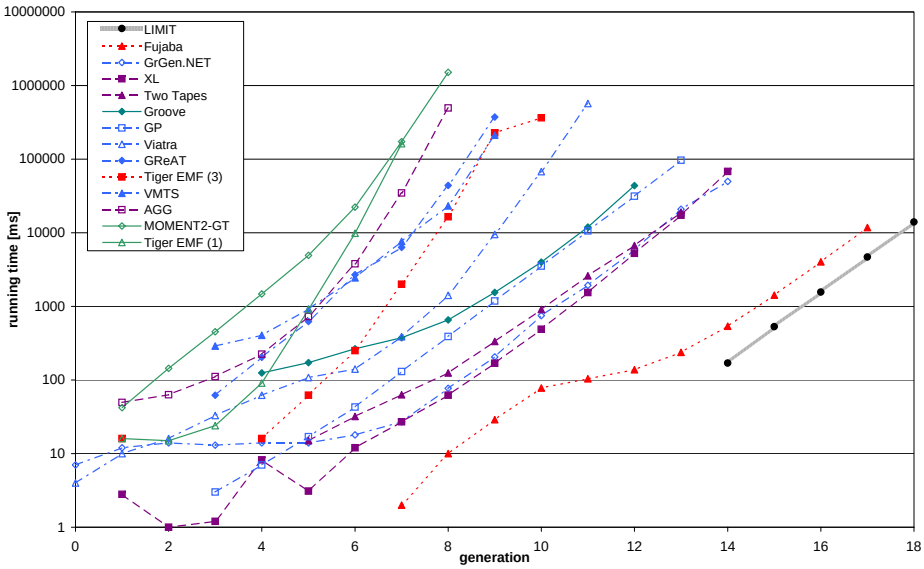


Fig. 15. Running times of the solutions shown in logarithmic scale. All measurements were carried out on different machines, so this figure has a deviation of a factor of about 3. The line style is selected according to the kind of matching and rule application of the solution: Kind 1 ⇒ solid line, kind 2 ⇒ dash-dot line, kind 3 ⇒ dotted line, and kind 4 ⇒ dashed line.

LIMIT is based on a compressed memory representation. Each node is represented by its (at most two) outgoing edges; the node itself uses no memory. The edges are stored in, and refer to, a single array. The Sierpinski triangles can be generated using only few types. Therefore, LIMIT uses a few bits of the

indices to encode the types of the respective nodes. This way edges can only be traversed in one direction with $O(1)$ the other direction possibly needs the inspection of the whole graph. The pattern matching is done by extracting just the right edges with the right direction from memory. The rewrite step just adds more nodes, i.e. edges.

5.3 Concluding Remarks

The generation of Sierpinski triangles is well suited to measure the memory footprint of a solution approach and therefore, well suited for tuning tools with respect to graph representation. It is pleasant to note, that there are tools capable of handling millions of nodes and edges in very little time; reasonable hand-coded solutions are only one order of magnitude faster and more memory efficient. Tools which allow custom graph representations can be well adapted to given problems which leads to usually better runtime performances than built-in representations. However, it would be preferable to deduce a very efficient host graph implementation directly from the meta model. Considering Table 1 again, we have to stress that the motivations for building graph transformation tools have been very diverse. Dependent on intended application domains, quality criteria such as performance, usability, correctness, validation facilities, etc. are considered with intensity of varying degree. Hence, only some tools allow for custom graph representations.

In the process of preparing contest solutions, a large part of the tool builders started to improve the performance of their tools. For several tools this case study has been the first application which creates huge graphs. That quickly showed that graph representations have to be very economic regarding memory. Some tool builders immediately started to reduce memory consumption significantly, others will follow in the next time. Improvements regarding time seem to be possible concerning the graph matching algorithms used. Often, simple patterns can be handled more efficiently than currently done.

The considered tools offer a wide range of features enabling developers to provide elegant solutions. However, although the case study is pretty small, missing features have been identified. E.g. the graphical layout of Sierpinski triangles was often not optimal. Furthermore, this case study led the interest especially to parallel matching and application of rules.

All solutions of kinds 1 and 2 allow some kind of non-determinism in the selection of matches and/or rules. However, this case study does not need any kind of non-determinism and Fig. 15 shows that solutions of kinds 3 and 4 tend to show a better performance. Due to this diversity of solution approaches, Figure 15 does not really provide an objective comparison of the tools' runtime performance. Note that many participants did provide more than one solution, in order to provide more elegant and more performant solutions. For more precise measurements, it would be preferable to choose one solution approach beforehand. Even if doing so, this case study does not measure well the matching time for patterns, since the patterns used are small. Moreover, the (non-deterministic) application

of many different rules and rule interaction are not considered. Hence, further case studies need to be considered in future.

References

1. Graphs for Object Oriented Verification. http://groove.cs.utwente.nl/.
2. Fujaba-Homepage. http://www.fujaba.de/, 2007.
3. Artur Boronat. The MOMENT2-GT web site, 2008. http://www.cs.le.ac.uk/people/aboronat/tools/moment2-gt.
4. Gernot Veit Batz, Moritz Kroll, and Rubino Geiß. A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In this volume.
5. B. Bohnet. Textgenerierung durch Transduktion linguistischer Strukturen. In *DISKI 298, AKA, Berlin*, 2006.
6. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude*. Springer LNCS Vol. 4350, 2007.
7. Eclipse Consortium. *Eclipse – Version 3.3*, 2007. http://www.eclipse.org.
8. Eclipse Consortium. *Eclipse Modeling Framework (EMF) – Version 2.3*, 2007. http://www.eclipse.org/emf.
9. VIATRA2 Framework. An Eclipse GMT Subproject. http://www.eclipse.org/gmt/.
10. Rubino Geiß and Moritz Kroll. GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In this volume.
11. Ole Kniemeyer and Winfried Kurth. The modelling platform GroIMP and the programming language XL. In this volume.
12. H.-J. Kreowski, R. Klempien-Hinrichs, and S. Kuske. Some essentials of graph transformation. In Z. Esik, C. Martin-Vide, and V. Mitrana, editors, *Recent Advances in Formal Languages and Applications. Vol. 25 of Studies in Computational Intelligence*, pages 229–254, Berlin Heidelberg New York, USA, 2006. Springer.
13. László Lengyel, Tihamér Levendovszky, Gergely Mezei, and Hassan Charaf. Control Flow Support in Metamodel-Based Model Transformation Frameworks. In *EUROCON 2005, Proceedings of the IEEE*, pages 595–598, Belgrade, Serbia and Montenegro, 2005.
14. Greg Manning and Detlef Plump. The GP programming system. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, Electronic Communications of the EASST, 2008. To appear.
15. Arend Rensink and Gabriele Taentzer. AGTIVE 2007 Graph Transformation Tool Contest. In this volume.
16. Tiger Developer Team. *Tiger EMF Transformer*, 2007. http://www.tfs.cs.tu-berlin.de/emftrans.
17. Tamás Vajk and Tihamér Levendovszky. Imperative OCL Compiler Support for Model Transformations. In *7th International Symposium of Hungarian Researchers on Computational Intelligence*, pages 166–178, Budapest, Hungary, November 2006.
18. VMTS website. http://vmts.aut.bme.hu/, 2007.

Appendix

Tool → Gen. ↓	LIMIT	Fujaba	GrGen .NET	XL	Two Tapes	GP	Groove	Viatra	VMTS	Tiger EMF (3)	GReAT	AGG	Tiger EMF (1)	MOMENT 2-GT
0			7					4						
1			12	3				10		16		50	16	42
2			14	1				16				63	15	144
3			13	1		3		33	290		62	111	24	453
4			14	8		7	125	62	406	16	204	224	91	1.475
5			14	3	15	17	172	108	920	62	624	729	906	4.928
6			18	12	32	43	265	141	2.450	250	2.688	3.788	9.870	22.284
7		2	27	27	63	130	375	388	7.630	2.000	6.344	34.724	162.427	172.868
8		10	77	62	125	390	656	1.413	23.220	16.500	43.906	495.059		1.511.668
9		29	206	169	334	1.183	1.547	9.474	211.750	227.250	373.781			
10		78	749	489	907	3.520	4.000	67.593		364.984				
11		104	1.930	1.542	2.600	10.619	11.969	568.589						
12		138	5.876	5.252	6.700	31.471	43.672							
13		238	20.872	17.321	18.900	96.474								
14	170	537	49.919	68.061										
15	531	1.417												
16	1.562	4.022												
17	4.687	11.778												
18	14.015													

Table 2. Comparison of the different approaches regarding the running time. Please note, that all measurements were conducted on different computers; hence only the tendency of the figures is significant (see Figure 15). To make the figures more accessible, we ordered the columns comparing their running times referring to the highest common available generation.