Developing Graph Transformations with GrGen.NET

Moritz Kroll and Rubino Geiß

Universität Karlsruhe (TH), 76131 Karlsruhe, Germany
{moritz|rubino}@ipd.info.uni-karlsruhe.de
http://www.grgen.net

Abstract. GrGen.NET is a graph rewrite system enabling elegant and convenient development of applications with comparable performance to conventionally developed ones. GrGen.NET uses attributed, typed, and directed multigraphs with multiple inheritance on node and edge types. Extensive graphical debugging integrated into an interactive shell complements the feature highlights of GrGen.NET. In this paper, the above claims are illustrated by examples.

1 What is GrGen.NET?

GrGen.NET is an application domain neutral graph rewrite system superseding GrGen [1]. This paper demonstrates the feature highlights of GrGen.NET regarding practical relevance:

Fully Featured Meta Model: Our tool uses attributed, typed, and directed multigraphs with multiple inheritance on node/edge types (see Section 3.1).

Expressive Rules, Fast Execution: The expressive and easy to learn rule specification language allows straightforward formulation of even complex problems (see Section 3.2) while remaining the fastest automatic graph rewrite system known to us (cf. [1, 2]).

Graphical Debugging: The GrShell offers interactive execution of rules visualising the current graph and the rewrite process (see Section 2).

2 Visualisation of Rule Applications

While developing graph rewrite applications, developers want to keep track of the changes made to the graph. For this purpose GrGen.NET provides an interactive command line shell called GrShell. This shell features a debug mode whereby stepwise execution of graph rewrite sequences and even a detailed visualisation of the rewrite process is possible. Graph rewrite sequences are rule compositions with logical, iterative, and transactional sequence control as well as variable assignments (see Section 4). We exemplify the visualisation of the rewrite process by a simple Koch snowflake [3]. A Koch snowflake is a fractal which starts with a triangle and replaces all edges as shown in Figure 1. The

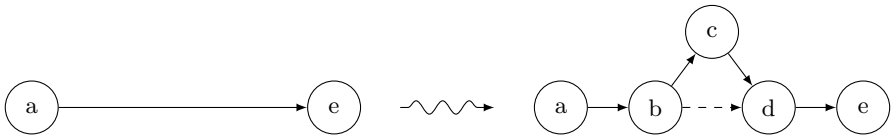


Fig. 1. The basic rule of the Koch snowflake

dashed line is included to help the graph layout system understand what we want to see visually.

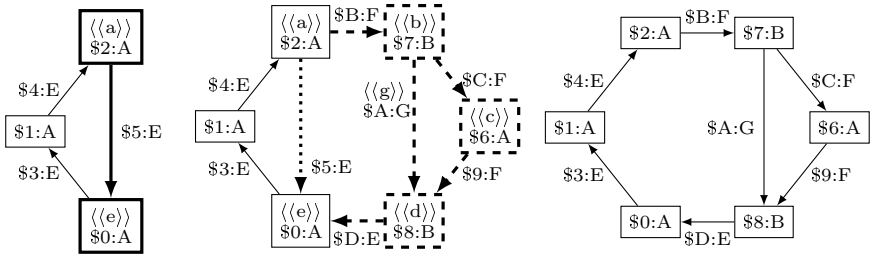
The listing below shows an implementation of the rule in Figure 1. `A` and `B` are node types inheriting from the root type `Node`; `E`, `F` and `G` are edge types inheriting from the root type `Edge`.

```
1 rule makeFlake1 {
2  pattern { a:Node -:E-> e:Node; }
3  replace { a -:F-> b:B -:F-> c:A -:F-> d:B -:F-> e;  b -g:G-> d; }
4 }
```

When we apply this rule to a graph containing a triangle in the debug mode of GrShell, the below steps will be visible. The mapping of the rule elements to the host graph is shown by a labelling with rule element names (e.g. ⟨⟨a⟩⟩).

1. The found match is marked – depicted by bold lines (see Figure 2(a)).
2. Created and deleted elements are highlighted – depicted by dashed and dotted lines, respectively (see Figure 2(b)).
3. The final result is shown (see Figure 2(c)).

This provides the user with a deep insight into the rewrite process, hinting at maybe erroneously omitted context of the pattern graph as well as incorrectly specified rewrite graphs. Moreover, this debugging feature is also useful for educational purpose. In the following we will have a look at an artificial example presenting a broad view of the features supporting elegant and efficient development.



(a) The match (b) The changes (c) The result

Fig. 2. Visualisation steps of the rewrite process

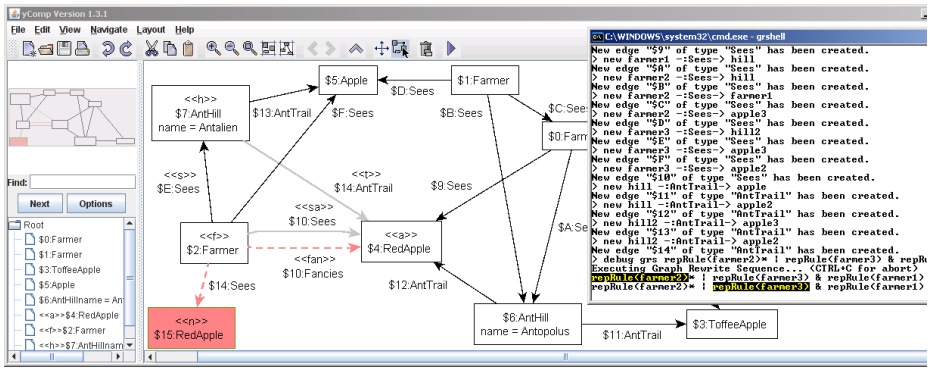


Fig. 3. Screenshot of GrShell and yComp in debug mode (colours optimised for print)

3 Showcase Example

Consider the small story given below. Along with its lines we developed a graph model (see Section 3.1) and a rewrite rule (see Section 3.2) for illustrating the most prominent features of GrGen.NET.

> A given farmer sees a special apple, but not a toffee apple. The farmer also sees an ant hill and notices that there is an ant trail between this hill and the apple. As the farmer is already angry, he looks around, whether there is any farmer (including himself in case he is reflective) seeing both him and the ant hill. When nobody watches he destroys the ant trail. Now the farmer fancies the apple and remembers it happily for further interactions. Due to quantum fluctuations a new apple of the same kind with more worms materializes out of the sudden.

Listing 1.1. A graph model

```
1  model AgriModel;
2  enum Country { NL=31, DE=49, NZ, GB }
3  node class Apple { weight : double; origin : Country; worms : int; }
4  abstract node class Red;
5  node class RedApple extends Apple, Red { tasty : boolean; }
6  node class ToffeeApple extends Apple;
7  node class AntHill { name : string; size : int; }
8  node class Farmer { angry : boolean; }
9  edge class AntTrail connect AntHill[2:5] -> Apple[*] { distance : float; }
10 edge class Sees connect Farmer[*] -> Apple[*],
11     Farmer[*] -> AntHill[*], Farmer[*] -> Farmer[*];
12 edge class Fancies extends Sees;
```

Listing 1.2. Rule with replace

```
1  actions AgriActions using AgriModel;
2  rule repRule(f:Farmer) : (Apple) {
3   pattern {
4    f -sa:Sees-> a:Apple\ToffeeApple
5     <-:AntTrail- h:AntHill <-s:Sees- f;
6    if { typeof(a) > Apple;
7         f.angry == true; }
8    negative { hom(f,o);
9     h <-:Sees- o:Farmer -:Sees-> f; }
10  }
11  replace {
12   a <-fan- f -s-> h;
13   -fan:Fancies<sa>->;
14   n:typeof(a) <-:Sees- f;
15   eval { f.angry = false;
16    n.worms = a.worms + h.size / 13;
17   }
18   return (a);
19  } }
```

Listing 1.3. Rule with modify

```
1  actions AgriActions using AgriModel;
2  rule modRule(f:Farmer) : (Apple) {
3   pattern {
4    f -sa:Sees-> a:Apple\ToffeeApple
5     <-t:AntTrail- h:AntHill <-:Sees- f;
6
7    // lines 6-9 same as repRule
8
9
10  }
11  modify {
12   delete(t);
13   -:Fancies<sa>->;
14
15
16   // lines 14-18 same as repRule
17
18
19  } }
```

3.1 Meta Model

Listing 1.1 shows a graph model for our showcase example.

Types: Nodes and edges have types. Types are vital for optimizing the matching process, so it is very important to put some effort into the development of a significant type system for a given problem.

Inheritance: Types can inherit from multiple other types (model line 5). This makes it easier to specify one rule for multiple similar patterns. Abstract types cannot be instantiated (model line 4).

Attributes: A graph element has attributes according to its type. The available attribute types are: `int`, `boolean`, `string`, `float`, `double`, and user-defined `enum` types. Node and edge types should be prefered in favour of attributes to get a significant type system.

Connection Assertions: A model can include restrictions on the "shape" of a graph by specifying degrees and types of nodes where an edge of a given type may connect to. Using these restrictions a graph can be validated on demand. An `AntHill` node must have two to five outgoing `AntTrail` edges to an `Apple` node and an `Apple` node may have any number of incoming `AntTrail` edges from `AntHill` nodes (model line 9).

3.2 Graph Rewrite Rules

Listings 1.2 and 1.3 show two equivalent implementations of our example using the model from Listing 1.1. Both rules differ in the way of specifying the rewrite

graph (rule line 11, explanation see below). Figure 3 shows an application of `repRule` in the debug mode of the GrShell visualized by our graph viewer yComp. In the SPO approach, graph rewrite rules are divided into two parts: The pattern graph L and the rewrite graph R connected by the preservation morphism r. In our rule language L and R are explicit via pattern and replace part. The morphism r is specified implicitly by the use of the same graph element names in both parts. The pattern part features:

Negative Application Conditions (NACs): Pattern graphs which must not be present in the host graph for the rule to be applicable are called NACs (rule line 8 and 9).

Attribute Conditions: These conditions specified inside an `if` clause must be fulfilled for a match (rule line 7). They may include arithmetic operations on attributes of multiple pattern elements.

Type Conditions: Types of pattern elements can be restricted with powerful type expressions. Even dynamic types of host graph elements may be used. Simple subtype exclusion can directly appear in type declarations of pattern elements (rule line 4), more complex conditions must be specified inside an `if` clause (rule line 6). The condition `SuperType` < `SubType` holds.

Isomorphic/Homomorphic Matching: By default matches are isomorphic, i.e. two pattern elements must not be mapped to the same graph elements of the host graph. Using the `hom` operator pattern elements can be declared to be matched homomorphic (rule line 8).

Parameter Passing: Rules can receive host graph elements from outside of their declarative domain. This allows hard to find elements to be passed to a rule avoiding unnecessary searching and non-determinism (see Section 4).

The rewrite part features:

Modify/Replace The rewrite graph can be specified in two modes (compare line 11 and 12 of both rules): Firstly, the replace mode in which the rewrite graph is specified completely. Secondly, the modify mode where only the modifications to the pattern graph are specified: Additions and changes are given straightforward, deletions are declared by the `delete` keyword. The error-prone repetition of graph elements can thereby be avoided. However if the preservation morphism r maps only a few elements, using the replace mode can be beneficial.

Attribute Re-calculation: Results of arithmetic expressions can be assigned to the attributes of matched and new elements (rule lines 15–17).

Retyping: Matched elements can be retyped – not neccessarily respecting the type hierarchy – keeping common attributes of initial and final type (rule line 13). This is important if a graph element must change its type but any unmatched context should remain untouched.

Creation with Dynamic Types: New graph elements can also be created using the actual types of matched elements (see line 14). This helps to avoid multiple similar rules.

Return Elements: Non-deleted elements can be returned for further computations, e.g. parameters for other rules (rule line 18).

4 Rule Application

Rule application is non-deterministic in two ways: The order of rule applications and the selection among multiple matches for a pattern graph. To cope with these kinds of non-determinism there are different possible strategies: Ranging from arbitrarily choosing one rule/match, over using some probability distributions, to oracular approaches that always choose the "right" alternative w.r.t. a certain goal. All these strategies can be implemented on top of the LIBGR programming interface. It provides direct access to the match and rewrite facility with full control over rule and match selection. Additionally GRGEN.NET has a high-level interface to programmed rule application available both from GRSHELL and LIBGR: Graph Rewrite Sequences (GRS). These sequences built of rules and sequence operators are composed by induction. Matches are chosen arbitrarily.

Logical: Let `op` be an Java-like Boolean operator: `&` (eager and), `^` (xor), | (eager or), `&&` (lazy and), || (lazy or). Consider the sequence $\texttt{s}_1$ `op` $\texttt{s}_2$. The operands $\texttt{s}_1$ and $\texttt{s}_2$ are executed according to the usual operator semantics. The success or failure of $\texttt{s}_1$ `op` $\texttt{s}_2$ is determined by its operands.

Iterative: The sequence `s*` executes until `s` fails, `s{n}` executes `n`-times.

Transactional: If the sequence `s` inside the nestable transactional brackets ⟨`s`⟩ fails, the changes introduced by the sequence `s` are undone. This allows backtracking on rule level at the expense of speed.

$-Operator-Prefix: Normally rules are chosen deterministically. `$` switches to probabilistic mode (equally distributed for both operands).

5 Conclusion

GRGEN.NET provides an expressive specification language including several features to simplify rule development. The GRSHELL allows for coarse-grained debugging of GRSs as well as fine-grained debugging of a single rule application. Beyond this paper Denninger demonstrated the practical usability by developing a compiler with GRGEN.NET [4]. Moreover, Batz et al. and Kroll show that our matching approach is both reasonable and fast [5, 2].

References

1. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: A fast SPO-based graph rewriting tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: ICGT. Volume 4178 of LNCS., Springer (2006) 383–397
2. Kroll, M.: GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen (2007) Studienarbeit, IPD Goos, Universität Karlsruhe.
3. Koch: Sur une courbe continue sans tangente obtenue pare une construction geometrique elementaire. Arkiv för matematik, astronomi och fysik **1** (1904) 681–702
4. Denninger, O.: Erweiterung des Kantenkonzepts deklarativer GES von Einfachkanten über Hyperkanten zu "Superkanten". Master's thesis, IPD Tichy (2007)
5. Batz, G.V., Geiß, R., Kroll, M.: A first experimental evaluation of search plan driven graph pattern matching (2007) Submitted to AGTIVE 2007.