Multiscale Graph Grammars Can Generate Cayley Graphs of Groups and Monoids

Winfried Kurth^(✉)

Universität Göttingen, Abteilung Ökoinformatik,
Biometrie und Waldwachstum, Büsgenweg 4,
37085 Göttingen, Germany
wk@informatik.uni-goettingen.de

Abstract. A graph grammar with parallel replacement of subgraphs, based on the single-pushout approach in graph rewriting, was designed which constructs Cayley graphs of monoids of transformations of a finite set, with permutation groups as a special case. As input, graph-based representations of a finite number of generating transformations have to be specified; they will then correspond to the edge types of the Cayley graph which is the final result of the rewriting process. The grammar has $7 + d$ rules, where d is the number of generators, and operates at two scale levels. The fine-scale level is the level of elements on which the transformations act and where their composition is calculated by parallel subgraph replacement. The coarse-scale level corresponds to the transformations themselves which are organized in the Cayley graph in a sequential rule application process. Both scale levels are represented in a single graph. The graph grammar was implemented in the programming language XL on the software platform GroIMP, a graph rewriting tool which was originally designed for simulating the growth of plants.

Keywords: Graph grammar · Cayley graph · Permutation groups · Transformation semigroups · GroIMP

1 Introduction

Cayley graphs have been used for a long time to visualize the structure of discrete groups (see, e.g., [2]). They can be generalized to other algebraic structures [1, 8]. Particularly, for a monoid M (i.e., semigroup with neutral element), the Cayley graph of M w.r.t. a set of generators G has the node set M , and for each node $n \in M$ and each generator $g \in G$ there is a directed edge $(n, n \circ g)$. In this small study we will demonstrate that the construction of the Cayley graph of a monoid which is defined by a generating set G of transformations of a finite set can be conceived as an application of a graph rewriting system. See [6] for an introduction to finite transformation semigroups. Permutation groups, which have been intensively studied for long time [3], and their Cayley graphs emerge as a special case when all members of G are bijective.

As our graph grammar formalism, we used the “relational growth grammars” (RGGs) which enable a parallel replacement of subgraphs and are based on the single-pushout approach [4]. “Parallel replacement” means here that in every discrete time step all subgraphs which are instances of a left-hand side of a grammar rule are replaced by the graph given on the respective right-hand side. (For a discussion of the advantages of the single-pushout approach see [9], p. 104.) RGGs are supported by the programming language XL and are available on the platform GroIMP [9, 10]. They can be seen as generalizations of Lindenmayer systems from strings to graphs and were primarily used to simulate the 3-dimensional architecture and development of plants (see, e.g., [14]). The software GroIMP contains also a powerful graph-drawing algorithm combining an energy-based layout with a tree layout [5]. Furthermore, XL was recently extended to support rewriting at several scale-levels simultaneously [12, 13]. These were our main reasons for choosing the RGG approach and GroIMP for our purpose. We think other graph-grammar formalisms and software tools with a suitable expressive power could solve our task as well.

It was not our intention to create yet another calculation tool for transformation semigroups or permutation groups. Several such tools, devoted to this special field of application, exist already (see, e.g., [11]). We rather wanted to show that an existing graph rewriting approach, originally introduced for a quite different purpose, can solve our construction task. Our method is not optimized in terms of computation time or memory efficiency.

In the following, we will specify our used graph model and all the rules of our grammar. The description of the rules will also contain the proof that they indeed build the Cayley graph defined by the generating transformations given as input. Finally we will discuss some weaknesses of our method and also the graph layout provided for the final results by GroIMP.

2 Method and Result

2.1 The Graph Model

The graphs supported by the language XL are finite, connected, directed, rooted graphs with attributed, typed nodes with inheritance and with typed edges [9]. That means, each node belongs to a node type, with a class hierarchy among the types like in object-oriented programming, and each node can optionally have parameters (attributes), their number depending on the node type. In our case, all parameters will be integers. There is always a single node of the distinguished type “Root”. Edges belong to one of a finite number of edge types (without hierarchy). There are no loops and no multiple edges of the same type and direction. XL provides three standard edge types called *successor* (*s*), *ramification* (*r*) and *decomposition* (*d*) which were taken from a graph model designed for multiscale descriptions of plant architecture in [7]. Beyond that, additional edge types can be defined by the user.

The concept of multiple scales within a graph has been introduced to represent several levels of spatial resolution in vegetation models [12]. Although there is no geometry and hence no “spatial” scaling in our application here, we retain

the concept of scale-level by denoting the transformations (i.e., the elements of our monoid; node type T) as “coarse-scale” and the n elements on which they operate (node type E), which form a finite set S , as “fine-scale”. Without loss of generality, we can assume $S = \{1, 2, \dots, n\}$. At the coarse-scale level we will additionally need *intermediate nodes* (type I) of indegree 1 and outdegree 1 between any connected pair of T nodes—this is particularly necessary in the case that the Cayley graph has loops, because our graph model does not permit loops, and by inserting an intermediate node each loop will be expanded to a cycle of length 2 –, and *connector nodes* (type C) which will provide the linkage to the fine-scale level.

The T nodes will have an index parameter which will help to control the step-by-step buildup of the Cayley graph. The I nodes will have a parameter i , initially 0, which indicates if the target T node of the corresponding edge was already used for constructing a subgraph at fine-scale level representing the transformation for which this T node stands ($i = 1$) and if this transformation was already compared, in a subsequent step, with all previously generated transformations ($i = 2$). The E nodes will have a parameter k between 1 and $2n$ indicating which element of S they represent, either as a preimage ($1 \leq k \leq n$) or as an image element ($n + 1 \leq k \leq 2n$) with respect to the transformation under consideration.

As edge types, we utilize the standard types s , r and d as well as one edge type g_i for each generator ($1 \leq i \leq d$), and additionally two edge types x and y which will be used at the fine-scale level to calculate the composition of two transformations. In our graph, a transformation $f : S \rightarrow S$ will be represented by a subgraph consisting of $2n$ nodes of type E , with edges of type x between nodes E_k and $E_{f(k)+n}$ where the subscript denotes the parameter of the node. Each element of S is thus represented twice. We could have used a single set of n nodes instead, with edges between E_k and $E_{f(k)}$, but because our graph rewriting mechanism does not allow non-injective embeddings of the left-hand side of a rule into the host graph, this would have made the calculation of the composition of two transformations more complicated.

2.2 The Grammar

Rule 1 (Initialization): From the start node (in XL called *Axiom*) a graph is created which corresponds to the identical transformation id , shown in Fig. 1. The nodes I , T and C represent the coarse-scale level. The C node is connected by decomposition edges with the E nodes at fine-scale level. The interconnections of the E nodes by edges of type x form the pattern of id on an n -element set. In XL, the initialization rule is coded as follows:

```
Axiom ==>> I(2) T(0) +> c:C /> E(1) -x-> E(n+1) </ c,
           for (int k = 2; k <= n; k++)
             ( c /> E(k) -x-> E(n+k) </ c );
```

where “+>” stands for a type r edge, a blank for a type s edge, “/>” for a type d edge, and “-x->” for a type x edge. The parenthesized part in the third line is iterated $n - 1$ times. The syntax “c:” in the first line introduces an identifier for

the particular instance of a node of type C generated at this place and allows to refer to this instance by using the label “ c ” in the rest of the rule.

Rule 2 (Copy): To build up the Cayley graph, the transformations which exist already in the graph have to be multiplied with the d generators. In the first step, we simply make d copies of a transformation which is already represented in our graph. (This rule will then be applied iteratively to all nodes of type T .) The mapping pattern at fine-scale level of a copied transformation is still identical with that of the original, i.e., the generators have not yet been applied (Fig. 2). This copy rule application is controlled by a global variable m , initially 0, which is incremented by 1 in each application. For the sake of simplicity, we display here only the code for the case $d = 2$, with generators g and h . The generalization to the general case, using the remainder of $m \bmod d$, is straightforward. On the right-hand side of the rule, the XL method `cloneSubgraph(c)` is used which creates a copy of the whole subgraph accessible from the node marked with label c :

```
t:T(i) +> c:C, (i == (int)(m/2)) ==>>
  if (m % 2 == 0)
    ( t [ c ] -g-> I(0) T(m+1) [ cloneSubgraph(c) ] )
  else
    ( t [ c ] -h-> I(0) T(m+1) [ cloneSubgraph(c) ] )
{m++;};
```

Fig. 1. The graph which is generated after the application of the initialization rule (rule 1). Edge types: s = thin unbroken, r = thick broken, d = thin dotted, x = thick unbroken. The fine-scale representation of the transformation id forms the bottom part of the graph, consisting of type E nodes.

Fig. 2. The copy rule for generator g_i . The type T node represents the transformation f . u_f is the subgraph at fine-scale level corresponding to f . On the right-hand side of the rule, the Cayley graph is extended by adding a g_i edge and the node representing $f \circ g_i$. The subgraphs at fine-scale level are not yet updated. Edge signatures are the same as in Fig. 1, with an additional g_i edge (thick unbroken arrow).

Rule Family 3 (Insertion): For each generator g_i , there is a rule which adds a third layer of type E nodes to the existing two layers. The new nodes are parameterized with $n + 1, n + 2, \dots, 2n$ like the nodes of the second layer (cf. bottom part of Fig. 1). Connections are established between the second and the third layer using edges of type y which mimic the application of the transformation g_i on S . These connection patterns are the input of our algorithm. The code of each of these d rules thus depends on the action of the corresponding generator. As an example, we show the rule code for the permutation $g = (3\ 4)$ on a two-element set. It makes use of a context condition, enclosed in starred parentheses, which ensures that the rule is only applied to T nodes with an incoming path consisting of a type g edge followed by a type s edge. The additional condition “`i.used == 0`” ensures that this insertion rule is applied only once to each transformation node in the Cayley graph:

```
(* T -g-> i:I T +> c:C /> *) r:E(j), (* c /> *) s:E(k),
(i.used == 0 && j == 3 && k == 4)
==>> r -y-> E(4) </ c, s -y-> E(3) </ c { i.used = 1; };
```

Note that in the third line the order of the parameters of the type E nodes, 4 and 3, has switched according to the represented transposition g which exchanges 3 and 4.

Rules 4 and 5 (Composition): Like the insertion rules, the two composition rules work exclusively at fine-scale level. They perform the composition of the two transformations represented between the first and the second layer and between the second and third layer of type E nodes, respectively. At the end, the subgraph where they have been applied has again only two layers of nodes, and the type y edges have disappeared (Fig. 3). Rule 4 is the proper composition rule.

Fig. 3. (a) Rule 4, (b) rule 5 for composing two transformations. Both rules are applied in parallel mode, but all applications of (a) have to be finished before the (b) applications start.

Rule 5 serves to remove unused E nodes of the second layer after rule 4 has already been applied. Rule 5 is superfluous in the case that all transformations are permutations, because then all layer-2 nodes will be consumed by the applications of rule 4.

In the XL code we have to ensure that rule 5 is only applied when all possible applications of rule 4 have already been done.

To this purpose, XL provides the method call “`derive()`” enforcing to carry out all preceding parallel rule applications listed before:

```
[ a:E -x-> E -y-> b:E ==>> a -x-> b; ]
derive();
[ E -y-> b:E ==> b; ]
```

Rule 6 (Redirect): After copying the subgraph representing an already existing transformation f , inserting the connection pattern for generator g_i at fine scale level and calculating the composition $f \circ g_i$ at this level, we have to check if this newly determined transformation occurs already in our graph. In this case, the corresponding type g_i edge at coarse-scale level which led to the T node under consideration has to be redirected to the existing T node (which has the identical pattern at fine-scale level below it), and the new T node, together with the attached subgraph below it, has to be deleted (Fig. 4). The search for identical subgraphs has to be done systematically, so in the XL code the rule application is embedded into a control structure with graph queries (syntactically marked by starred parentheses). The flag “used” guarantees that the redirect rule is applied only once to a transformation node.

```
for ((* i:I y:T +> c:C *))
  if (i[used] == 1) {
    for ((* j:I z:T +> d:C *))
      if ((j[used] == 2) && equalTransf(c, d))
        { [ i y ==>> i z; ] }
    i[used] = 2;  }
```

The method “equalTransf”, called in line 5 of this code, checks if the fine-scale subgraphs below the nodes c and d represent the same transformation (code not shown).

Fig. 4. The redirect rule (rule 6). The italicized numbers in parentheses show the values of the parameter “used” of the type I nodes, which ensure that the rule is applied only once to each newly created type T node (here: the T node below the I node with value 1). The rule is applied only if the condition $(u = u')$ is valid for the indicated subgraphs. Edge signatures as in Fig. 1.

Global control. After the growing Cayley graph was (possibly) modified by the redirect rule, the rule application process starts again with the copy rule. The indexing of the type T nodes and the application condition in the copy rule ensure that each transformation is processed only as often as there are generators (i.e., d times). In XL, the correct order of rule applications is specified by a control

structure where the rule blocks are called by their respective names (“compose” stands for rules 4 and 5):

```
for (int i = 1; i <= d; i++) {
    copy(); derive();
    insert_gi(); derive();
    compose(); derive();
    redirect(); derive();
}
```

The execution of this loop of rule applications has itself to be iterated until there are no more applicable rules (i.e., all transformations have been processed). The result is then a graph consisting of the Cayley graph of the monoid generated by $g_1, \dots, g_d$ at coarse-scale level (however, including insertion and connector nodes) and with all generated transformation patterns at fine-scale level. As an example, Fig. 5 shows the generated graph in the case of the Klein four-group $Z_2 \times Z_2$.

Fig. 5. The full graph representing the Klein four-group, generated by the permutations (1 2) and (3 4). The Cayley graph is spanned by the four nodes of type T (shaded). Edge signatures as in Figs. 1 and 2, additionally thick dotted arrows for the second generator.

Rules 7 and 8 (Simplification): To obtain the proper Cayley graph, we can first get rid of the fine-scale level by applying the rule $C \implies \epsilon$; which deletes all connector nodes. Since in the single-pushout approach, all “dangling edges” are deleted after removing a part of the host graph by rule application, this disconnects all type E nodes from the root and thus, because of the requirement of connectedness, deletes them, too. Only the coarse-level part of the generated graph (without the C nodes) will be retained after this (parallel) rule application. Finally, we can simplify the two-edge paths between two transformation nodes by merging them into one edge: $t:T \xrightarrow{-gi-} I \xrightarrow{u:T} t' \xrightarrow{-gi-} u$; for all

generators, $i = 1, \dots, d$. This last step should only be done if the Cayley graph has no loops (which is always the case for groups).

3 Discussion

The presented graph-rewriting method follows closely the way how one would construct the Cayley graph of a given transformation monoid manually. To this end, a combination of parallel subgraph replacement rule applications and procedural control structures (for ensuring the systematic, sequential visiting of all type T nodes during the construction process, and for comparing their transformations with all previously generated ones) turned out to be useful. The language XL provides an appropriate environment for combining these programming paradigms. However, some steps still required workarounds. Particularly, for testing the identity of two (graphically-encoded) transformations we had to write our own method “equalTransf” for using it in rule 6. The copying method “cloneSubgraph”, used in rule 2, was already provided by XL, but, as for equality checking, a more intuitive and short-hand notation would have made the rules even more transparent.

In terms of calculation time, our method is far from being optimized. Particularly the search pattern in the insertion rule family tends to produce a lot of mismatches which are then sorted out during testing the additional parameter conditions, especially for larger values of n . A more intelligent pattern matching strategy for the transformation-encoding subgraphs would probably be possible. On the other hand, our calculation of the composition of transformations by parallel graph-grammar rule application (rule 4) is potentially very efficient. To make use of this parallelism, however, requires a splitting of the execution into several threads and their processing by properly parallel devices (e.g., on the GPU).

Cayley graphs are used to display an algebraic structure graphically and thereby to visualize its symmetries (see, e.g., [2] for groups). This requires an appropriate graph layout algorithm which is able to map abstract symmetries into geometrical ones. GroIMP provides already a “standard” graph layout which is a combination of an energy-based layout (using simulated annealing for optimization) and a classical tree layout [5]. When applied to the full graph (before simplification), it has the advantage to automatically separate the fine-scale from the coarse-scale level. From the examples of final Cayley graphs (after simplification) which we tested with our algorithm, we could see that this layout tends to produce already quite appropriate results in the case of groups, whereas in the more general case of monoids the existing patterns of similarity in the Cayley graphs are often not fully reflected in the layouts. It will be a special challenge in graph drawing to provide an improved graph layout algorithm for Cayley graphs of finite monoids which could then be used in combination with our graph-rewriting approach.

Acknowledgments. I thank Ole Kniemeyer and Octave Etard for their ingenious programming work. Some prerequisites of this study were developed in the project

“Multiscale functional-structural plant modelling at the example of apple trees”, funded by DFG under grant number KU 847/11-1. All support is gratefully acknowledged.

References

1. Caucal, D.: On Cayley graphs of basic algebraic structures. arXiv preprint [arXiv: 1903.06521](https://arxiv.org/abs/1903.06521)
2. Coxeter, H.S.M., Moser, W.O.J.: Generators and Relations for Discrete Groups, 4th edn. Springer, Berlin (1980). <https://doi.org/10.1007/978-3-662-21946-1>
3. Dixon, J.D., Mortimer, B.: Permutation Groups. Springer, Heidelberg (1996). <https://doi.org/10.1007/978-1-4612-0731-3>
4. Ehrig, H., et al.: Algebraic approaches to graph transformation II: single pushout approach and comparison with double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 1: Foundations, pp. 247–312. World Scientific, Singapore (1997)
5. Etard, O.: General-purpose graph drawing algorithm for GroIMP. Internship work presentation, University of Göttingen (2011, Unpublished)
6. Ganyushkin, O., Mazorchuk, V.: Classical Finite Transformation Semigroups: An Introduction. Springer, London (2009). <https://doi.org/10.1007/978-1-84800-281-4>
7. Godin, C., Caraglio, Y.: A multiscale model of plant topological structures. J. Theor. Biol. **191**, 1–46 (1998)
8. Kelarev, A.V., Praeger, C.E.: On transitive Cayley graphs of groups and semigroups. Eur. J. Comb. **24**, 59–72 (2003). [https://doi.org/10.1016/S0195-6698\(02\)00120-8](https://doi.org/10.1016/S0195-6698(02)00120-8)
9. Kniemeyer, O.: Design and implementation of a graph grammar based language for functional-structural plant modelling. Ph.D. thesis, University of Technology at Cottbus (2008). <http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:kobv:col-opus-5937>
10. Kurth, W., Kniemeyer, O., Buck-Sorlin, G.: Relational growth grammars – a graph rewriting approach to dynamical systems with a dynamical structure. In: Banâtre, J.-P., Fradet, P., Giavitto, J.-L., Michel, O. (eds.) UPP 2004. LNCS, vol. 3566, pp. 56–72. Springer, Heidelberg (2005). https://doi.org/10.1007/11527800_5
11. Linton, S.A., Pfeiffer, G., Robertson, E.F., Ruškuc, N.: Computing transformation semigroups. J. Symb. Comput. **33**, 145–162 (2002). <https://doi.org/10.1006/jasco.2000.0406>
12. Ong, Y.: Multi-scale rule-based graph transformation using the programming language XL. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 417–419. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_29
13. Ong, Y., Streit, K., Henke, M., Kurth, W.: An approach to multiscale modelling with graph grammars. Ann. Bot. **114**, 813–827 (2014). <https://doi.org/10.1093/aob/mcu155>
14. Smoleňová, K., Kurth, W., Cournède, P.-H.: Parallel graph grammars with instantiation rules allow efficient structural factorization of virtual vegetation. Electron. Commun. EASST **61**, 1–17 (2013). <https://doi.org/10.14279/tuj.eceasst.61.830.825>