Electronic Communications of the EASST
Volume 61 (2013)

Selected Revised Papers from the
4th International Workshop on
Graph Computation Models
(GCM 2012)

Parallel Graph Grammars with Instantiation Rules Allow Efficient Structural Factorization of Virtual Vegetation

Katarína Smoleňová,Winfried Kurth, and Paul-Henry Cournède

17 pages

Guest Editors: Rachid Echahed, Annegret Habel, Mohamed Mosbah
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Parallel Graph Grammars with Instantiation Rules Allow Efficient Structural Factorization of Virtual Vegetation

Katarína Smoleňová[1],Winfried Kurth[2], and Paul-Henry Cournède[3]

[1] ksmolen@gwdg.de
[2] wk@informatik.uni-goettingen.de
Abteilung Ökoinformatik, Biometrie und Waldwachstum
Georg-August-Universität Göttingen, Germany

[3] paul-henry.cournede@ecp.fr
Laboratoire Mathématiques Appliquées aux Systèmes
École Centrale Paris, France

Abstract: Parallel rewriting of typed attributed graphs, based on the single-pushout approach extended by connection transformations, serves as the backbone of the multi-paradigm language XL, which is widely used in functional-structural plant modelling. XL allows to define instantiation rules, which enable an instancing of graphs at runtime for frequently occurring substructures, e.g., in 3-d models of botanical trees. This helps to save computer memory during complex simulations of vegetation structure. Instantiation rules can be called recursively and with references to graph nodes, thus providing a unifying formal framework for various concepts from the literature: object instancing, structural factorization, Xfrog multiplier nodes, L-systems with interpretation. We give simple examples and measure the computation time for an idealized growing virtual plant, taken from the GreenLab model, in its implementation with instantiation rules in XL, compared to a version without instantiation rules.

Keywords: instantiation rules, structural factorization, XL, growth grammar

1 Introduction

Realistic simulation of large vegetation scenes or growing plant structure may become computationally very intensive if no methods to reduce the geometrical complexity are used. Especially in the case of tree growth simulations, the number of organs may fast exceed several millions.

To save computer memory, different techniques have already been proposed in the field of computer graphics. When employing one of them, object instancing [Sut63], the geometric representation is specified just for one or more representative plants (master). Other identical plants (instances) are placed at different positions in the scene, having a reference to the master object. Instances may be exact copies of the referenced object [Har92] or approximate ones [DHL+98, Bro96].

Thanks to the repetitive nature of plants, instancing may also be applied on several levels within the plant structure. To efficiently render trees that look realistic from both distant and close views Max [Max96] modelled larger structures from smaller ones: a leaf is used to form a twig with several leaves, several twigs form a big twig, big twigs form a subbranch, and so on.

Cournède proposed a mathematical formalism [CKM+06] for decomposition of a growing plant into substructures [RGQH03], that strongly relies on botanical principles. It assumes that all organs of the same kind, created at the same time, behave identically. Identical organs, or rather groups of organs (substructures), are then treated as instances. This formalism of "structural factorization" of plants is of special interest in the area of functional-structural plant modelling, where plant structure and growth is modelled as the result of underlying physiological processes governed by environment. These models are calibrated on real measurements, and to identify the model parameters, the simulation process has to be performed a large number of times [CKM+06]. The algorithm is used in the GreenLab model [CKM+06, MCL+09] and allows efficient computation of both, plant development (deterministic and stochastic) and functioning. In combination with parallel computing, it was also used for fast simulations of forest growth [CGB+09].

In higher plants, the elementary botanical units named metamers or phytomers are set in place rhythmically or continuously depending on the type of plants [BC07]. In the rhythmic case, the plant grows by successive shoots of several metamers produced by buds. The appearance of these shoots defines the architectural growth cycle. A growth unit is the set of metamers built by a bud during a growth cycle. Growth units can be of different kinds and ordered according to botanical rules, like acrotony. Plant growth is said continuous when meristems keep on functioning and generate metamers one by one. The number of metamers on a given axis (that is to say generated by the same meristem) is generally proportional to the sum of daily temperatures (above a reference temperature depending on the species) received by the plant. In such case, the growth cycle is defined as the thermal time unit necessary for a meristem to build a new metamer. In both continuous and rhythmic cases, the chronological age of a plant (or of an organ) is defined as the number of growth cycles it has existed for, and the organogenesis is used as the time step to model the plant as a discrete dynamical system.

Quite naturally, owing to the simultaneous functioning of the meristems, parallel string rewriting grammars, i.e. L-systems [PL90], are most widely used to model plant organogenesis. The possibility to factorize L-system productions was pointed out already by Smith [Smi84]. Hart [Har92] and Brownbill [Bro96] studied methods for converting L-system models into object instancing models. In [Har92], also recursive instancing was allowed. Using the idea of substructure instances [YBRH02], Ding *et al.* [DZZ06] demonstrated an improvement of the efficiency of an L-system-based algorithm for creation of simple trees. If only organogenesis is modelled, the mathematical formalism of structural factorization can also be expressed by L-systems [CKM+06].

Widely used in the area of functional-structural plant modelling is the language XL (eXtended L-system language) [Kni08]. XL is a superset of the language Java and supports the specification of graph grammars. It also allows to define instantiation rules, which enable an instancing of graphs for frequently occuring substructures, e.g., in 3-d models of botanical trees. Instantiation rules were successfully used to implement multiplication components for modelling of plants as used in the modelling software Xfrog [Hen06]. In this paper we would like to explore their use for implementing the formalism of structural factorization.

We first give a theoretical background on the language XL, including the formal definition of instantiation rules. Afterwards we show how to use these rules to produce inflorescence patterns, fractals, and to efficiently simulate plant development based on structural factorization.

2 Theoretical Background

The graph model which describes the basic data structure of the language XL is that of *typed attributed graphs with inheritance over a type graph* [Kni08] (p. 98). "Typed" means that nodes of the graph belong to a finite number of classes, called "modules" in XL, and in an application of a graph grammar rule, the types of matching nodes have to coincide. Likewise, the edges belong to a finite number of edge types, which have also to be taken into account during graph matching. For a given edge type, at most one edge of this type is allowed between two given nodes. The edge types can thus be interpreted as *relations* on the set of nodes. A "type graph" can be specified to restrict the allowed types of edges between nodes of given types. "Inheritance" is a concept taken from object-oriented programming and allows a hierarchy among the node types. "Attributed" refers to the nodes which can have a finite number of attributes of arbitrary data type.

The parallel application of a *graph grammar* on a graph conforming to this model is in [Kni08] (p. 101ff.) defined as a special case of the *single-pushout approach* from algebraic graph grammar theory [EHK+97] extended by connection transformations and operators. The latter provide an embedding mechanism for the right-hand sides of rules into the host graph which allows to consider classical parallel string-rewriting rules (L-system rules) as a special case of graph rewriting.

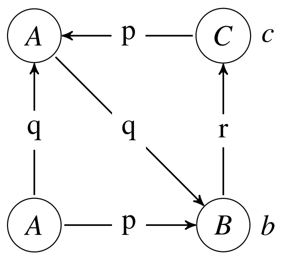
In the following, we simplify the definitions from [Kni08] in order to avoid too much technical overhead. Particularly, we omit the attributes, the inheritance relation and the restriction imposed by a type graph. "Instantiation rules" as a special sort of graph transformation rules have not been formally defined in [Kni08], although they have been described as part of the language XL there and have been used in various examples.

Definition 1 A Relational Growth Grammar graph (*RGG graph*) is a quintuple $G = (T_N, T_E, N, E, t)$, where $T_N \neq \emptyset$ is a finite set of node types, $T_E \neq \emptyset$ a finite set of edge types, N a set of nodes, $E \subseteq N \times T_E \times N$ a set of typed edges, and $t : N \to T_N$ a node typing function. An *RGG graph isomorphism* is a graph isomorphism which preserves node and edge types.

The language XL makes use of a string notation of RGG graphs which is demonstrated in Figure 1.

Definition 2 A *bi-rooted RGG graph* (G, n_0, n_1) is an RGG graph with two distinguished nodes $n_0, n_1 \in N$.

Definition 3 An (unconditional) *RGG rule* with L-system style embedding is a pair $((G_1, m_0, m_1), (G_2, n_0, n_1))$ of two bi-rooted RGG graphs (G_1, m_0, m_1) (the *left-hand side* or *search pattern*) and (G_2, n_0, n_1) (the *right-hand side* or *production pattern*). The *application* of an RGG rule to a host RGG graph H consists of the simultaneous replacement of all subgraphs of H which are isomorphic (via an RGG graph isomorphism) to G_1 by G_2, with a subsequent redefinition of



```
A [ -p-> b:B -r-> c:C ] -q-> A [ -q-> b ] <-p- c
```

Figure 1: Example of an RGG graph G with node types A, B, C and edge types p, q, r. Given is also a string notation $s(G)$ of the graph in XL (not unique). b and c are labels referring to particular nodes.

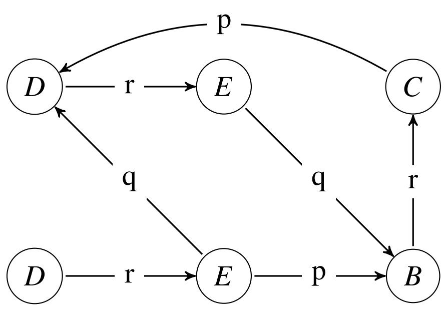


Figure 2: Result of a parallel rule application to the RGG graph of Figure 1 (see text).

some of the incoming and outgoing edges: Each edge going from the rest of the host graph to the node m_0 in G_1 is redirected to n_0, and each edge going from m_1 in G_1 to the rest of the host graph gets n_1 as its new start node. All other edges going into or out of G_1 are deleted.

In the case that an edge goes from m_1 in one isomorphic copy of G_1 to m_0 in another copy, an edge of the same type is inserted from n_1 of the first copy of G_2 to n_0 of the second copy of G_2 (see [Kni08] for a more rigorous definition). In the case of overlapping copies of G_1 in H, the result of rule application is undefined.

For shortness, we write `G1 ==> G2` for an RGG rule, omitting the nodes m_0, m_1, n_0 and n_1. In the language XL, the distinguished nodes of an RGG rule are not given explicitly, but are taken from the string representation of the graphs: m_0 is the leftmost and m_1 the rightmost node in $s(G_1)$, and analogously for G_2. For instance, the application of the RGG rule

```
A ==> D -r-> E
```

to the graph in Figure 1 will result in the graph depicted in Figure 2.

The advantage of this form of embedding the right-hand side into the host graph is its consistency with parallel string rewriting by L-systems. A string which is transformed by an L-system can simply be represented by a linear chain of nodes, connected by edges of just one special type, which we call “successor edges”. The symbols of the L-system correspond to the node types.

Note that our definition of an RGG rule does not require that the l.h.s. G_1 and the r.h.s. G_2 are disjoint. This allows to preserve individual nodes of G_1, standing for objects in the applications, with all their attributes during rule application. In XL, this preservation is enforced by node labels which serve as unique identifiers. E.g., after an application of the rule

```
a:A -p-> B ==> a -q-> B,
```

the type `A` node named `a` from the l.h.s. is preserved, while the type `B` node is deleted and replaced by another type `B` node.

Definition 4 An *RGG* (Relational Growth Grammar) (T_N, T_E, G_0, R) consists of a finite set of node types $T_N \neq \emptyset$, a finite set of edge types $T_E \neq \emptyset$, a start RGG graph (axiom) G_0 and a finite set of RGG rules R, with G_0 and R having all node and edge types from T_N, resp., T_E.

The parallel application of rules from an RGG to a host graph is performed in the same way as in the case of a single rule. In XL, all node types in T_N have to be declared first, using the keyword "module" (also possible: "class"). XL allows to split the set of rules into several subsets which are accessible under user-defined names. Control structures like conditional branches or loops can be used. Likewise, these control structures can also occur on the r.h.s. of rules to control the production of subgraphs. We omit these extensions in our definitions because they are not central for the notion of instantiation rules.

Definition 5 An *instantiation rule* is an RGG rule $N ==> G$, where the l.h.s. consists only of a single node of type N. An *RGG with instantiation rules* is a quintuple (T_N, T_E, G_0, R, I) where the first 4 components form an RGG and I is a finite set of instantiation rules with node and edge types from T_N, resp., T_E.

The application of an RGG with instantiation rules is performed as shown in Figure 3. $G_1, G_2, \ldots$ are the RGG graphs which are successively obtained from the start graph G_0 by parallel application of (potentially) all rules from R in each transformation step. They usually correspond to rough structural descriptions of the simulated objects at discrete timesteps. $S_0, S_1, \ldots$ are the RGG graphs which are obtained from $G_0, G_1, \ldots$ by parallel application of (potentially) all rules from I. These rules can also be applied recursively. The graphs S_i usually stand for detailed geometrical descriptions (scene graphs) of the simulated objects. $S_0, S_1, \ldots$ can be regarded as a simulated developmental sequence of the objects. Note that this way of applying two sets of rules is not equivalent to that used in "table L-systems", since the graphs S_i do not undergo a new application of rules in the next step. Nondeterminism is possible for both rule sets, R and I, when it makes sense in an application, and can be resolved in XL by stochastic choice of rules with given probabilities. When XL is used, the graphs S_i are not stored as data structures in the computer memory, but are derived "on the fly" by applying rules from I to G_i at runtime in the moment when they are needed (e.g., for rendering an image of the simulated structures). In the case of large graphs, this can lead to a gain in efficiency.

The same scheme of rule application is also valid for L-systems or graph grammars with *interpretive rules* [Kni08] (also called *interpretative* [Kur94] (p. 25), *interpretation* [PHM00], or *homomorphism rules* [Měc05]). Interpretive rules differ from instantiation rules in the feature that multiple instances of a right-hand side of an interpretive rule which are obtained from G_i

$$\begin{tikzcd}
G_0 \arrow[r, "R"] \arrow[d, "I"'] & G_1 \arrow[r, "R"] \arrow[d, "I"'] & G_2 \arrow[r, "R"] \arrow[d, "I"'] & \cdots \arrow[d, phantom, "\cdots"] \\
S_0 & S_1 & S_2 & \cdots
\end{tikzcd}$$

Figure 3: Way of operation of an RGG with normal RGG rules R and instantiation rules I, starting from RGG graph G_0.

are stored as different copies (being parts of S_i) and can thus later be accessed and manipulated individually. This is not possible for the results of instantiation rule application. The latter are, however, less demanding in terms of computation space and time.

A further extension of the concept of instantiation rule allows to construct graphs which encode in a concise manner the way how objects or groups of objects are multiplied and transformed in space, including recursive constructions leading to fractals. As special cases, the multiplicator nodes used in the interactive plant modelling software Xfrog [DL05] can be derived this way [Hen06].

Definition 6 An *RGG with instantiation rules and references* is an RGG where the right-hand sides of the instantiation rules can contain specific nodes of the form $Ref(e)$, where e is an edge type. When applied to a node a in the host graph H, each node of form $Ref(e)$ is replaced by a copy of the subgraph of H which is directly connected with a via an edge of type e emerging from a, if such an edge exists (otherwise the $Ref(e)$ node is deleted). In case of more than one edge of type e starting in a, the choice of the adjacent subgraph is nondeterministic.

For instance, the instantiation rule

```
R ==> Ref(p) -q-> Ref(p),
```

applied to the host graph `R -p-> A`, will result in the graph `A -q-> A -p-> A`. Since in most applications, the edges of type `p` will be ignored during further processing (e.g., rendering - only the subgraph spanned by a special subset of edge types is used for rendering in XL), we will effectively have a copy of the referenced `A` node: `A -q-> A`. This can be extended to more complex copy operations, as we will see in the examples sections. In the language XL, the `Ref` nodes are effectively realized by calls of the method `getFirst(e)`.

3 Examples

3.1 Inflorescence Patterns and Fractals

We will first show two simple examples, providing the complete code in the language XL, which demonstrate that the concepts introduced above can be expressed in XL in a straightforward way. The first example (code listed in Table 1) defines a node "`Infl`" (standing for "inflorescence") by an instantiation rule with references. Similar to the effect of the "Hydra" node from Xfrog

Table 1: Listing of the XL code for the inflorescence example (see text).

```
 1 const int m = EDGE_0;
 2 module Infl ==> for (int i: 1:250) ([
 3    { float h = i * 0.02;
 4      float s = 0.2 * Math.sqrt(i); }
 5    M(-h) RH(i*137.5) Translate(s,0,0) RU(i*80/250)
 6    Scale(0.2,0.2,0.3*h+0.1) getFirst(m)
 7 ]);
 8 public void run() [
 9    Axiom ==> P(10) Cylinder(20, 1) Cone(5.2, 2.4) P(14)
10              Infl -m-> Sphere;
11 ]
```

[DL05], the execution of this rule creates 250 scaled copies of the structure attached via an edge of type *m* and arranges them in a pattern following spiral phyllotaxis (Figure 4, right-hand part; see [Hen06] for the emulation of the other Xfrog arrangement nodes in XL).

In XL, edges of the standard type "successor" are automatically generated when nodes are separated by blanks (or linefeeds), as in the right-hand side of the rule in line 9. An edge type other than the standard ones requires a declaration of a corresponding identifier (here "`m`", done in line 1) and can then be used in the form "`-m->`" (line 10), here to connect the node of type `Infl` with a node corresponding to a geometrical object, here a `Sphere` with default radius 1. `Sphere` belongs, together with `Cylinder`, `Cone` and the geometrical transformation node types `M` (move), `Translate` (shift by a vector), `RH` (rotate around head direction), `RU` (rotate around up direction), `P` (set colour index) and `Scale` (rescale coordinates for the subsequent scene subgraph) to the default node types of XL, which do not need to be declared in the code, whereas `Infl` is a user-defined node type declared with an instantiation rule in lines 2-7. The "`for`" control structure in line 2 (with its body ending in line 7) iterates the production of the nodes which are specified in lines 5-6, together with the execution of the imperative statements in lines 3-4, with a counting index `i` running from 1 to 250. The calculations of `h` and `s` and the rotation angles and scaling factors in lines 5-6 were fitted to the botanical object to be modelled. The user can edit these lines to adapt the structure of the created inflorescence to empirical findings - and has thus much more flexibility than with a given standard portfolio of multiplier nodes as provided by the Xfrog software [DL05].

From the default start node called "Axiom", the execution of the "`run`" method (line 8) generates in one step of rule application a chain of four nodes with geometrical/visible meaning (line 9), followed by a node of type `Infl` and – attached by an `m` edge – a single `Sphere` node (line 10). This graph is saved internally (Figure 4 left).

During rendering, the `Infl` node is further transformed by the instantiation rule: In each iteration of the loop (line 2), a separate branch (enclosed by brackets [ ], line 2 and 7) is created, consisting of geometry nodes (line 5) and a scaled copy of the sphere from line 10, which is found by following the `getFirst` reference (line 6) along the edge `-m->` from the processed

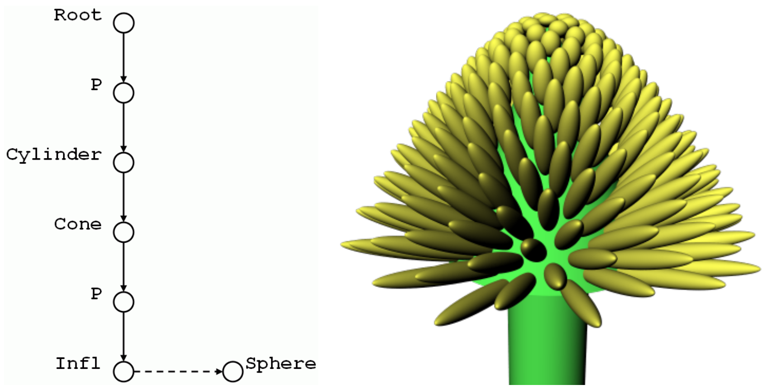


Figure 4: Structure resulting from the inflorescence example (Table 1). Left: Generated graph after the application of the single non-instantiation rule (lines 9-10 in Table 1). Unbroken edges stand for the successor relation, broken edges for a type m connection. Right: Rendered view of the corresponding 3-d structure, taking the application of the instantiation rule (lines 2-7) into account.

`Infl` node in line 10. `getFirst(m)` corresponds to a reference node $Ref(m)$. This results in the visual arrangement of 250 ellipsoids shown in Figure 4 (right part).

In an analogous way, the code from Table 2 produces after n steps of application of the `run` rules (lines 8-9) a chain of $n-1$ nodes of type `Rec` and one node of type `A` (Figure 5, left part). The instantiation of the `Rec` nodes by the translated copies which are referenced in the instantiation rule in lines 3-6 renders this graph into a Sierpinski triangle of recursion depth n (Figure 5, right part). Note that the stored graph is growing only linearly in n. The initiator `A` is here getting its shape from a sphere (line 2), but the shape of the initiator has no influence on the resulting fractal pattern for large n. The method can easily be generalized to other iterated function systems (IFS) or recurrent IFS (RIFS), cf. [Har92].

The Sierpinski triangle can also be generated by an instantiation rule in a purely recursive way, without references (Table 3). The generated graph is then even simpler: it consists of only two nodes, `Root` $\rightarrow$ `Rec`, and its structure does not change during rule application. Only the parameter n of the `Rec` node, denoting recursion depth, is incremented. A disadvantage is that the graph carries less visible information for the user.

Both recursive versions of the XL code for the Sierpinski fractal are not optimal in terms of efficiency, since in an implementation on a real computer the size of the recursion stack will soon be limiting when n is increased. However, XL provides also the possibility to implement the rewriting rule of the Sierpinski triangle in a direct way (without instantiation), avoiding heavy stack loading, as was shown in [TBB$^+$08] (p. 528). This solution turned out to be quite efficient and allowed to reach a recursion depth of 20, resulting in a graph with more than 5 million nodes.

Table 2: XL code for Sierpinski triangle, version with instantiation rules with references.

```
const int m = EDGE_0;
module A extends Sphere(0.4);
module Rec ==>
   Scale(0.5) [ Translate(0, 0.5, 0)        getFirst(m) ]
              [ Translate(-0.433, -0.25, 0) getFirst(m) ]
              [ Translate(0.433, -0.25, 0)  getFirst(m) ];
public void run() [
   Axiom ==> A;
   A ==> Rec -m-> A;
]
```

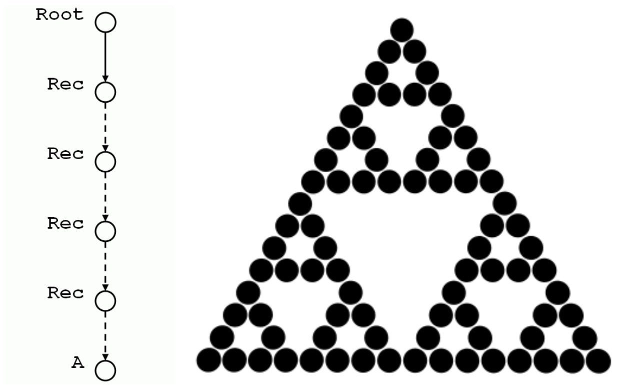


Figure 5: Structure resulting from the Sierpinski triangle example (Table 2). Left: Generated graph, right: rendered view of the corresponding 3-d structure.

Table 3: XL code for the Sierpinski triangle, recursive version without references (output as in Figure 5, right-hand side).

```
module Rec(int n) ==>
   if (n == 0) (
      Sphere(0.4)
   ) else (
      Scale(0.5) [ Translate(0, 0.5, 0)        Rec(n-1) ]
                 [ Translate(-0.433, -0.25, 0) Rec(n-1) ]
                 [ Translate(0.433, -0.25, 0)  Rec(n-1) ] );
public void run() [
   Axiom ==> Rec(0);
   Rec(n) ==> Rec(n+1);
]
```

3.2 Structural Factorization of Plant Architecture

In an individual plant, cohorts of similar organs are created at each growth cycle. Many simulation models handle each of them individually, which may lead to cumbersome computation in the case of tree growth simulations, as the number of organs may exceed several millions. However, it is often not necessary to consider local environmental conditions at the organ level. Thus, we suppose that all organs of the same kind, created at the same growth cycle, behave identically. From a modelling point of view, this leads to a powerful structural factorization of the plant, based on botanical instantiations: organs are grouped by categories, for example based on their physiological ages, characterizing their morphogenetic differentiation, see [BC07]. Compact inductive equations of organogenesis can thus be deduced, as detailed in [CKM+06].

Let P be the maximum number of metamer categories that we consider in the plant. It is generally very small, inferior to 5. At growth cycle t, a metamer is characterized by its physiological age p, the physiological age of its axillary branches q (with $q \geq p$) and its chronological age n. It will be denoted by $m^t_{pq}(n)$. These three indices p, q, n are sufficient to describe all the metamers and their numbers grow linearly with t. A bud is only characterized by its physiological age p and will be denoted by s_p.

The terminal bud of a plant axis produces different kinds of metamers bearing axillary buds of various physiological ages. These buds themselves give birth to axillary branches and so on. A substructure is the complete plant structure that is generated after one or several cycles by a bud. In the deterministic case, all the substructures with the same physiological and chronological ages are identical if they were set in place at the same moment in the tree architecture. At cycle t, a substructure is thus characterized by its physiological age p and its chronological age n. It will be denoted by $S^t_p(n)$. Since the physiological age of the main trunk is 1, at growth cycle t, the substructure of physiological age 1 and of chronological age t, $S^t_1(t)$, represents the whole plant. The total number of different substructures in a plant of chronological age t is small, usually less than 30, even if the total number of organs is high. Substructures and metamers are repeated a lot of times in the tree architecture, but they need to be computed only once for each kind.

We use the concatenation operator (represented by the product symbol) to describe the organization of plant metamers and substructures and deduce their construction at growth cycle t by induction, as follows:

- Substructures of chronological age 0 are buds:

$$S^t_p(0) = s_p$$

- If we suppose that they built all substructures of chronological age $n-1$, we deduce the substructures of chronological age n

$$S^t_p(n) = \left[\prod_{p \leq q \leq P} \left(m^t_{pq}(n)\right)^{u_{pq}(t+1-n)} \left(S^t_q(n-1)\right)^{b_{pq}(t+1-n)} \right] S^t_p(n-1) \tag{1}$$

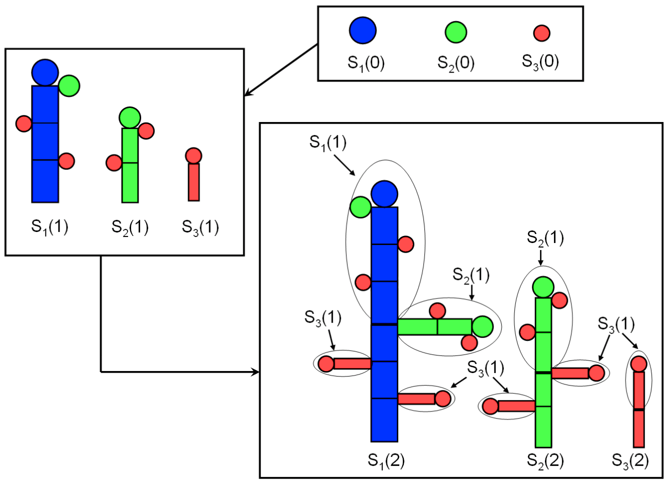


Figure 6: Construction of substructures for a plant with deterministic development. The substructure of physiological age 1 and chronological age 2, $S_1(2)$, is built of the base growth unit, consisting of two metamers of type m_{13} and one metamer of type m_{12}, and of substructures of chronological age 1 (created in the previous step): two lateral substructures of physiological age 3, one lateral substructure of physiological age 2, and the terminal substructure of physiological age 1.

For all (p,q) such that $1 \le p \le P$, $p \le q \le P$, $\left(u_{pq}(t)\right)_t$ and $\left(b_{pq}(t)\right)_t$ are sequences of integers that are characteristic of the plant organogenesis: $u_{pq}(t)$ corresponds to the number of metamers m_{pq} in growth units of physiological age p appearing at growth cycle t; $b_{pq}(t)$ is the number of axillary substructures of physiological age q in growth units of physiological age p that appeared at growth cycle t. These sequences can be deterministic (fixed or determined by the functional part of the plant as detailed in [MCL+09]) or stochastic (the induction equation (1) would be generalized in such case with the generating functions of the number of elements in plant architecture, see [LC08]).

In Equation 1, substructure $S^t_p(n)$ is decomposed into:

- its oldest growth unit, called base growth unit:

$$\prod_{p \le q \le P} \left(m^t_{pq}(n)\right)^{u_{pq}(t+1-n)}$$

- the lateral substructures borne by the base growth unit (they are one cycle younger):

$$\prod_{p \le q \le P} \left(S^t_q(n-1)\right)^{b_{pq}(t+1-n)}$$

- the substructure grown from the apical bud of the base growth unit (also one cycle younger):

$$S^t_p(n-1)$$

Table 4: XL code for plant development based on structural factorization, recursive version with an instantiation rule.

```
module Bud(int p) extends Sphere
{ ... /* set radius, shader */ }
module Metamer(int p, int q) extends Cylinder
{ ... /* set radius, length, shader */ }
module Substructure(int p, int n) ==>
   if (n > 0) (
      for (int j = 5; j >= p-1; j--) (
         for (int i = 0; i < u[p-1][j]; i++) (
            RH(ang_ph[p-1]) Metamer(p, j+1)
            for (int k = 0; k < b[p-1][j]; k++) ( [
               if (k > 0) ( RH(360.0 / b[p-1][j] * k) )
               RU(ang_br[p-1][j]) Substructure(j+1, n-1)
            ] )
         )
      )
      Substructure(p, n-1)
   ) else (
      Bud(p)
   );
public void run() [
   Axiom ==> Substructure(1, 0);
   Substructure(p, n) ==> Substructure(p, n+1);
]
```

Note that for deterministic growth, the substructures are independent of growth cycle t. The decomposition of a plant with deterministic development is illustrated in Figure 6. If we append geometrical rules (e.g., internode lengths, branching angles, phyllotaxy) to the structural equations, we will obtain the 3-d architecture of a geometrical tree.

The (deterministic) construction of substructures can be directly translated to XL code, using instantiation, as shown in Table 4. The phyllotactic and branching angles are part of the instantiation rule, with parameters `ang_ph` and `ang_br`, respectively. The arrays `u`, `b` stand for u_{pq}, resp. b_{pq}, with values set as in [KRB$^+$03]. The resulting structure is shown in Figure 7 (middle, right). The graph generated after the rule applications consists, similarly to the purely recursive Sierpinski triangle example, of two nodes, `Root -> Substructure`.

We have modified the recursive algorithm from Table 4 to store the substructures of all physiological and chronological ages (as shown in Figure 6) in a list, so that they could be reused as instances later on. An example of the underlying graph is shown in Figure 7 (left).

Furthermore, our basic algorithm for plant development, not taking advantage of structural factorization and instantiation rules, was presented in [SHK12].

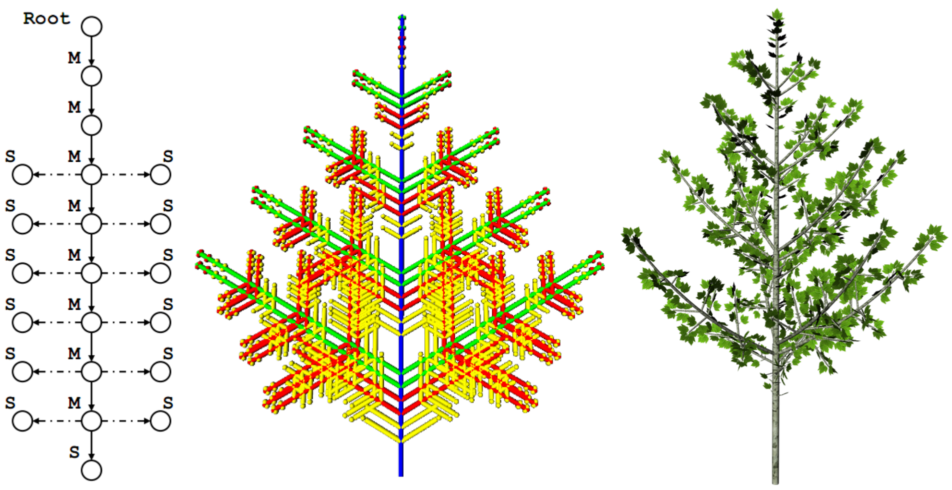


Figure 7: Model example. Left: The essential structure of the generated graph (non-recursive version) for the substructure of physiological age 1 and chronological age higher than 0. It consists of metamers (M) of the base unit and corresponding substructures (S). Lateral substructures are connected to metamers by a branching edge (dash-dot style). Middle: Simulated topology at growth cycle 5 for $S_1(5)$. Right: The same topological structure with improved geometrical representation.

To compare the computation cost of structure generation in the three above-mentioned implementations of plant development, simulation time for the development of the same virtual plant (Figure 7 middle, cf. [KRB+03, SHK12]) as well as memory requirements were measured on a computer with Intel Core i7 CPU 950, 3.07 GHz, and 12 GB RAM. We show the results in Table 5 and Figure 8 for six age stages. At the last stage, represented by age 30, the number of metamers in the plant structure reaches over 4 million. Together with buds, the total number of objects in the simulated structure is over 4.3 million. Without instantiation rules, the computation time for the simulation is about 103 seconds. With instantiation rules, the time is reduced by factor of 2,000 if the algorithm with lists is used, and even more if the recursive algorithm is used. Also the memory requirements are reduced; to 5% or 2%, respectively. Note that the measured time corresponds to the execution of the growth rules in the method `run`. To obtain the final geometry if instantiation rules are used, e.g., for rendering, the graph has to be traversed (the instantiation rules are executed).

4 Conclusions

Object instancing is an established technique improving the performance when modelling and rendering vegetation. Giving the theoretical background first, we presented the features of the XL language, supporting instancing of graphs, i.e., instantiation rules. The language XL with its instantiation rules provides a unifying formal framework for concepts of object instancing, Xfrog multiplier components, structural factorization, L-systems with interpretation. Here, we demon-

Table 5: Simulation time (in ms) and memory requirement (in MB) for the growing plant from Figure 7. A (Basic) refers to the basic algorithm without structural factorization from [SHK12], B (Recursion) refers to the recursive algorithm from Table 4, and C (List) shows the performance of the algorithm that stores the substructures in a list. Simulation time (Time A, B, C) corresponds to the execution of the growth rules. To measure the total time (with final geometry) if instantiation rules are used, the time for the graph traversal has to be added. See text for more details.

Plant age	5	10	15	20	25	30
Metamers	2,440	44,480	238,120	775,360	1,928,200	4,048,640
Objects	3,621	54,441	272,461	857,681	2,090,101	4,329,721
Time A (Basic)	30.96	618.61	3,915.22	15,421.00	40,540.11	103,633.07
Time B (Recursion)	0.29	0.59	0.83	1.06	1.39	1.71
Time C (List)	6.55	14.39	23.17	31.38	41.39	50.63
Graph traversal B	6.47	63.06	301.68	957.49	2,391.31	4,578.85
Graph traversal C	9.21	81.61	391.00	1,192.62	2,739.19	5,370.88
Total time B	6.76	63.65	302.51	958.55	2,392.70	4,580.56
Total time C	15.76	96.00	414.17	1,224.00	2,780.58	5,421.51
Time ratio C/B	22.6	24.4	27.9	29.6	29.8	29.6
Time ratio A/C	4.7	43.0	169.0	491.4	979.5	2,046.9
Total time ratio A/B	4.6	9.8	12.9	16.1	16.9	22.6
Total time ratio A/C	2.0	6.4	9.5	12.6	14.6	19.1
Memory A	32	42	155	341	791	1573
Memory B/A (%)	88	69	17	6	3	2
Memory C/A (%)	247	183	54	23	10	5

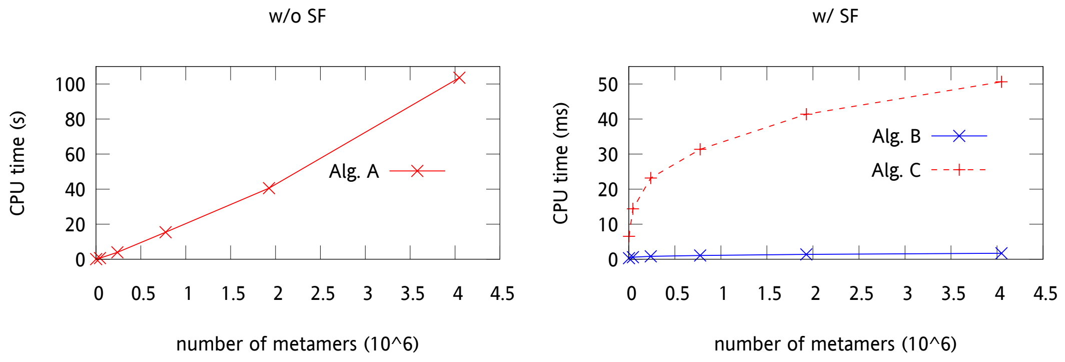


Figure 8: Simulation time for the growing plant from Figure 7 (for algorithms A, B, C according to Table 5).

strated their use for modelling inflorescence patterns and fractals, and finally for the structural factorization of deterministically growing plants, decreasing the computational cost of simulations greatly.

Further work may comprise several issues. We would like to explore which algorithm, implementing structural factorization, is most suitable for modelling stochastic structures and physiological processes within plants. Although the recursive algorithm gives the best results for the tested structure in terms of computation time and memory, it might carry too less information for further use. The proposed instantiation rule for generating a plant structure is very simple. By varying the input parameter values it can be used to generate various plant species. The rule can easily be extended to support bud mutation (for explanation of the term see [RGQH03]), too. Certainly, the ability to factorize a plant structure, i.e. to specify the parameter values for the topology, depends on the experience of the user. Another direction of the future work would be to investigate how to automatically factorize existing L-system plant models (based on [SS01]) while minimizing artefacts in visual appearance and functioning caused by possible reduction of geometry. Important factor that should be kept in mind is the botanical fidelity of the models.

Acknowledgements: We thank Ole Kniemeyer, Reinhard Hemmerling and Michael Henke for helpful comments. This research was funded by the German Research Foundation (DFG) under project identifier KU 847/8-1.

Bibliography

[BC07] D. Barthélémy, Y. Caraglio. Plant Architecture: A Dynamic, Multilevel and Comprehensive Approach of Plant Form, Structure and Ontogeny. *Annals of Botany* 99(3):375–407, Mar. 2007.

[Bro96] A. Brownbill. Reducing the Storage Required to Render L-system Based Models. Master's thesis, University of Calgary, 1996.

[CGB+09] P.-H. Cournède, T. Guyard, B. Bayol, S. Griffon, F. Coligny, P. Borianne, M. Jaeger, P. de Reffye. A Forest Growth Simulator Based on Functional-Structural Modelling of Individual Trees. In *Proc. IEEE 3rd International Symposium on Plant Growth Modeling, Simulation, Visualization and Applications*. Pp. 34–41. IEEE Computer Society Press, Los Alamitos, 2009.

[CKM+06] P.-H. Cournède, M.-Z. Kang, A. Mathieu, J.-F. Barczi, H.-P. Yan, B.-G. Hu, P. de Reffye. Structural Factorization of Plants to Compute Their Functional and Architectural Growth. *Simulation* 82(7):427–438, July 2006.

[DHL+98] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, P. Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proc. 25th Annual Conference on Computer Graphics and Interactive Techniques*. Pp. 275–286. ACM, New York, 1998.

[DL05] O. Deussen, B. Lintermann. *Digital Design of Nature: Computer Generated Plants and Organics*. Springer-Verlag, Berlin Heidelberg, 2005.

[DZZ06] W.-L. Ding, W.-T. Zhang, X. Zhou. An Improved Algorithm Based on Sub-Structures for Creating Virtual Plant. In *Proc. 16th IEEE International Conference on Artificial Reality and Telexistence–Workshops*. Pp. 200–204. Computer Society Press, Los Alamitos, 2006.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini. Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. Pp. 247–312. World Scientific, River Edge, 1997.

[Har92] J. C. Hart. The Object Instancing Paradigm for Linear Fractal Modeling. In *Proc. Conference on Graphics Interface*. Pp. 224–231. San Francisco, Morgan Kaufmann Publishers Inc., 1992.

[Hen06] M. Henke. Entwurf und Implementation eines Baukastens zur 3D-Pflanzen-visualisierung in GroIMP mittels Instanzierungsregeln. Master's thesis, BTU Cottbus, 2006.

[Kni08] O. Kniemeyer. *Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling*. PhD thesis, BTU Cottbus, 2008.

[KRB+03] M.-Z. Kang, P. de Reffye, J.-F. Barczi, B.-G. Hu, F. Houllier. Stochastic 3D tree simulation using substructure instancing. In *Proc. International Symposium on Plant Growth Modeling, Simulation, Visualization and their Applications*. Pp. 154–168. Springer / Tsinghua University Press, Beijing, 2003.

[Kur94] W. Kurth. *Growth Grammar Interpreter GROGRA 2.4: A software tool for the 3-dimensional interpretation of stochastic, sensitive growth grammars in the context of plant modelling. Introduction and reference manual*. B 38. Ber. FZW Göttingen, 1994.

[LC08] C. Loi, P.-H. Cournède. Generating functions of stochastic L-systems and application to models of plant development. In *Proc. Fifth Colloquium on Mathematics and Computer Science*. Pp. 325–338. 2008.

[Max96] N. L. Max. Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers. In *Proc. Eurographics Workshop on Rendering Techniques*. Pp. 165–174. Springer-Verlag, Wien, 1996.

[MCL+09] A. Mathieu, P.-H. Cournède, V. Letort, D. Barthélémy, P. de Reffye. A dynamic model of plant growth with interactions between development and functional mechanisms to study plant structural plasticity related to trophic competition. *Annals of Botany* 103(8):1173–1186, June 2009.

[Měc05] R. Měch. CPFG Version 4.0 User's Manual. 2005. http://algorithmicbotany.org/lstudio/CPFGman.pdf

[PHM00] P. Prusinkiewicz, J. Hanan, R. Měch. An L-system-based plant modeling language. In Nagl et al. (eds.), *AGTIVE'99*. LNCS 1779, pp. 395–410. Springer-Verlag, Berlin, Heidelberg, 2000.

[PL90] P. Prusinkiewicz, A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990.

[RGQH03] P. de Reffye, M. Goursat, J. P. Quadrat, B.-G. Hu. The Dynamic Equations of the Tree Morphogenesis GreenLab Model. In *Proc. International Symposium on Plant Growth Modeling, Simulation, Visualization and their Applications*. Pp. 108–117. Springer / Tsinghua University Press, Beijing, 2003.

[SHK12] K. Smoleňová, M. Henke, W. Kurth. Rule-based integration of GreenLab into GroIMP with GUI aided parameter input. In *Proc. IEEE 4th International Symposium on Plant Growth Modeling, Simulation, Visualization and Applications*. Pp. 347–354. IEEE, Beijing, 2012.

[Smi84] A. R. Smith. Plants, fractals, and formal languages. *Computer Graphics* 18(3):1–10, July 1984.

[SS01] R. Schultz, H. Schumann. Automatic Instancing of Hierarchically Organized Objects. In *Proc. 17th Spring conference on Computer graphics*. Pp. 63–70. IEEE Computer Society, Washington, DC, 2001.

[Sut63] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proc. Spring Joint Computer Conference*. Pp. 329–346. ACM, New York, 1963.

[TBB$^+$08] G. Taentzer, E. Biermann, D. Bisztray, B. Bohnet, I. Boneva, A. Boronat, L. Geiger, R. Geiß, A. Horvath, O. Kniemeyer, T. Mens, B. Ness, D. Plump, T. Vajk. Generation of Sierpinski triangles: A case study for graph transformation tools. In Schürr and A. Zündorf (eds.), *AGTIVE'07*. LNCS 5088, pp. 514–539. Springer-Verlag, Berlin, Heidelberg, 2008.

[YBRH02] H.-P. Yan, J.-F. Barczi, P. de Reffye, B.-G. Hu. Fast algorithms of plant computation based on substructure instances. *Journal of WSCG* 10(3):SH–145, 2002.