Functional-Structural Plant Modelling with GroIMP and XL

Tutorial and Workshop at Agrocampus Ouest, Angers, 5-7 May, 2015

Winfried Kurth

University of Göttingen, Department Ecoinformatics, Biometrics and Forest Growth

Graph rewriting, interpretive rules, instantiation rules
The formal background of the programming language XL:
Relational Growth Grammars (RGG)
= a special form of parallel graph grammars

see

[http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:kobv:co1-opus-5937]
The step towards graph grammars

Drawback of L-systems:

- in L-systems with branches (by turtle commands) only 2 possible relations between objects: "direct successor" and "branch"

extensions:

- to permit additional types of relations
- to permit cycles

→ graph grammar
a string:
a very simple graph

- a string can be interpreted as a 1-dimensional graph with only one type of edges
- successor edges (successor relation)
in GroIMP, all is represented in a graph:

(Smoleňová 2010)
to make graphs dynamic, i.e., to let them change over time:

**graph grammars**

example rule:
A relational growth grammar (RGG) (special type of graph grammar) contains:

- an alphabet
  - the definition of all allowed
    - node types
    - edge types (types of relations)
- the axiom
  - an initial graph, composed of elements of the alphabet
- a set of graph replacement rules.
How an RGG rule is applied

- each left-hand side of a rule describes a subgraph (a pattern of nodes and edges, which is looked for in the whole graph), which is replaced when the rule is applied.

- each right-hand side of a rule defines a new subgraph which is inserted as substitute for the removed subgraph.
Example:

rule:

application:
a complete RGG rule can have 5 parts:

(* context *), left-hand side, ( condition )

==> 
right-hand side { imperative XL code }
in text form we write (user-defined) edges as

- edgetype->

edges of the special type "successor" are usually written as a blank (instead of -successor->)

also possible:  >

Further special edge types with special notation:
"branch" edge:  +> (also generated after "[")
"decomposition" edge:  />
L-systems as a special case of graph grammars:

- the symbols of the L-system alphabet become vertices
- concatenation of symbols corresponds to *successor* edges

example: graph grammar in XL for the Koch curve

```java
public void derivation()
{
    [Axiom => RU(90) F(10);
     F(x) => F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3);]
```

vertex of the graph edge (type „successor“)
a “proper“ graph grammar (not expressible as L-system):

rule in text form:  \[ i \rightarrow b \rightarrow j \rightarrow a \rightarrow k \rightarrow a \rightarrow i \rightarrow j \]
a “proper” graph grammar (not expressible as L-system):

rule:

application:

what happens if there are two nodes on the right-hand side instead of one?
2 types of rules for graph replacement in XL:

- **L-system rule**, symbol: $\Rightarrow$
  provides an *embedding* of the right-hand side into the graph (i.e., incoming and outgoing edges are maintained)

- **SPO rule**, symbol: $\Rightarrow\Rightarrow$
  incoming and outgoing edges are *deleted* (if their maintenance is not explicitly prescribed in the rule)

„SPO“ from „single pushout“ – a notion from universal algebra
example:

\[
\begin{align*}
\text{a:A} & \implies \text{a C} & \text{(SPO rule)} \\
\text{B} & \implies \text{D E} & \text{(L-system rules)} \\
\text{C} & \implies \text{A} \\
\end{align*}
\]

start graph:

\[\begin{array}{c}
\text{A} \rightarrow \text{B} \rightarrow \text{C}
\end{array}\]
\[ a:A \implies a C \]  
\[ B \implies D E \]  
\[ C \implies A \]  

(SPO rule)  
(L-system rules)
a: A ==> a C  
B  ==>  D E  
C  ==>  A

(SPO rule)
(L-system rules)
a: \( A \rightarrow a \ C \)  
\( B \rightarrow D \ E \) 
\( C \rightarrow A \)  

(SPO rule)  
L-system rules)  

= final result
test the example  *sm09_e27.rgg* :

```java
module A extends Sphere(3);

protected void init()
[    Axiom ==> F(20, 4) A; ]

public void runL()
[    A ==> RU(20) F(20, 4) A;
]

public void runSPO()
[    A ==>^ RU(20) F(20, 4, 5) A;
]

(^ denotes the root node in the current graph)
```
Representation of graphs in XL

- vertex types must be declared with "module"
- vertices can be all Java objects
- notation for vertices in a graph:
  Node_type, optionally preceded by: label:
  Examples: A, Meristem(t), b:Bud
- notation for edges in a graph:
  –edgetype–> (forward), <-edgetype– (backward),
  –edgetype– forward or backward,
  <-edgetype–> forward and backward
- special edge types:
  successor edge: –successor–>, > or (blank)
  branch edge: –branch–>, +> or [ refinement edge: />
Notations for special edge types

>       successor edge forward
<       successor edge backward

---- successor edge forward or backward

+>       branch edge forward
++       branch edge backward

--+-       branch edge forward or backward

/>       refinement edge forward

</>       refinement edge backward

---+       arbitrary edge forward

<---      arbitrary edge backward

--       arbitrary edge forward or backward

(cf. Kniemeyer 2008, p. 150 and 403)
### Notations for special edge types (overview)

<table>
<thead>
<tr>
<th></th>
<th>forward</th>
<th>backward</th>
<th>forward or backward</th>
<th>forward and backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>successor</td>
<td>&gt;</td>
<td>&lt;</td>
<td>− − −</td>
<td>&lt; − &gt;</td>
</tr>
<tr>
<td>branch</td>
<td>+ &gt;</td>
<td>&lt; +</td>
<td>− + −</td>
<td>&lt; + &gt;</td>
</tr>
<tr>
<td>refinement</td>
<td>/ &gt;</td>
<td>&lt; /</td>
<td>− / −</td>
<td>&lt; / &gt;</td>
</tr>
<tr>
<td>arbitrary</td>
<td>− − &gt;</td>
<td>&lt; − −</td>
<td>− −</td>
<td>&lt; − − &gt;</td>
</tr>
</tbody>
</table>
user-defined edge types

const int xxx = EDGE_0;  // oder EDGE_1, ..., EDGE_14
...


Notation of graphs in XL

example:

is represented in programme code as

a:A [-e-> B C] [<-f- D] -g-> E [a]

(the representation is not unique!)

(  >: successor edge,  +: branch edge)
how can the following graph be described in XL code?

*(the solution is not unique)*
**Interpretive rules**

insertion of a further phase of rule application directly preceding graphical interpretation (without effect on the next generation)
Example:

```java
module Stem extends Cylinder(3, 0.1)
{
    { setShader(GREEN); }
}

module Flower;

protected void init()
[
    Axiom ==> Stem
    Flower
;
    { applyInterpretation(); }
]

protected void interpret()
[
    Flower ==> 
    for ((1:5)) ( 
        RH(72) [ RL(80) Parallelogram(1, 0.5). (setShader(RED)) ]
    ) 
    Sphere(0.15). (setShader(YELLOW))
;
]```
Each occurrence of the interpreted vertex (here: Flower) is individually represented in the graph.

A special (internal) edge type and special vertices are used to link the interpretation results with the rest of the graph:
further example:

```java
public void run()
{
    [ 
        Axiom ==> A;
        A ==> Scale(0.3333) for (int i:(-1:1))
            for (int j:(-1:1))
                if ((i+1)*(j+1) != 1)
                    ( [ Translate(i, j, 0) A ] );
    ]
    applyInterpretation();
}

public void interpret()
{
    [ 
        A ==> Box;
    ]
}
```

generates the so-called „Menger sponge“ (a fractal)
public void run() {
    [ 
        Axiom ==> A;
        A ==> Scale(0.3333) for (int i:(-1:1))
            for (int j:(-1:1))
                if ((i+1)*(j+1) != 1)
                    ( [ Translate(i, j, 0) A ] );
    ]
    applyInterpretation();
}

public void interpret() {
    [ 
        A ==> Box;
    ]

    (a) A ==> Sphere(0.5);

    (b) A ==> Box(0.1, 0.5, 0.1)

    (c) A ==> Box(0.1, 0.5, 0.1)
        Translate(0.1, 0.25, 0) Sphere(0.2);
what is generated by this example?

```java
public void run()
{
    
    [Axiom ==> [ A(0, 0.5) D(0.7) F(60) ] A(0, 6) F(100);
    A(t, speed) ==> A(t+1, speed);
    ]
    applyInterpretation();
}

public void interpret()
{
    [
    A(t, speed) ==> RU(speed*t);
    ]
}
a very similar type of rules in XL: 

*instantiation rules*

purpose: replacement of single modules by more complicated structures, only for visual representation (similar as for interpretive rules)

- but: less data are stored (less usage of memory)
- only one vertex in the graph for the instantiated structure
- in contrast to interpretive rules, no turtle commands with effect on other nodes can be used

further, arising possibility: “replicator nodes“ for copying and relocation of whole structures
instantiation rules: syntax

no new sort of rule arrow

specification of the instantiation rule directly in the declaration of the module which is to be replaced

```
module A ==> B C D;
```

replaces (instantializes) everywhere A by B C D
the flower example again:

```plaintext
module Stem extends Cylinder(3, 0.1) {
    { setShader(GREEN); }
}

module Flower
==> for (((1:5))) {
    RH(72)[ RL(80) Parallelogram(1, 0.5).(setShader(RED)) ]
    Sphere(0.15).(setShader(YELLOW))
}

protected void init()
[
    Axiom ==> Stem
    Stem
    Flower
];
```
the resulting graph:
another example:

Usage of instantiation rules for multiplyer objects

sm09_e43.rgg

```cpp
const int multiply = EDGE_0;    /* user-defined edge type */

module Johnny => F(20, 1)
    [ M(-8) RU(45) F(6, 0.8) Sphere(1) ]
    [ M(-5) RU(-45) F(4, 0.6) Sphere(1) ] Sphere(2);
```

Johnny is instantiated with the red structure
another example:

Usage of instantiation rules for multiplyer objects

sm09_e43.rgg

```cpp
const int multiply = EDGE_0; /* user-defined edge type */

module Johnny ==> F(20, 1)
              [ M(-8) RU(45) F(6, 0.8) Sphere(1) ]
              [ M(-5) RU(-45) F(4, 0.6) Sphere(1) ] Sphere(2);

module Replicator ==> [ getFirst(multiply) ] Translate(10, 0, 0)
                      [ getFirst(multiply) ];
```

Johnny is instantiated with the red structure

inserts all what comes after the „multiply“ edge
another example:

Usage of instantiation rules for multiplier objects

```
const int multiply = EDGE_0;    /* user-defined edge type */

module Johnny ==> F(20, 1)
    [ M(-8) RU(45) F(6, 0.8) Sphere(1) ]
    [ M(-5) RU(-45) F(4, 0.6) Sphere(1) ] Sphere(2);

module Replicator ==> [ getFirst(multiply) ] Translate(10, 0, 0)
    [ getFirst(multiply) ];

public void run()
    [ Axiom ==> F(2, 6) P(10) Replicator -multiply-> Johnny;
```

Johnny is instantiated with the red structure

inserts all what comes after the „multiply“ edge
result:
Example: Inflorescence architecture

XL code

```java
const int m = EDGE_0;
module Infl => for (int i: 1:250) [
    { float h = i * 0.02;
        float s = 0.2 * Math.sqrt(i);
    }
    M(-h) RH(i*137.5) Translate(s,0,0) RU(i*80/250)
    Scale(0.2,0.2,0.3*h+0.1) getFirst(m)
]
public void run() [
    Axiom => P(10) Cylinder(20, 1) Cone(5.2, 2.4) P(14)
    Infl -m-> Sphere;
]```
Example: Inflorescence architecture
generated graph and 3-d result
Example: Inflorescence architecture

Frangipani example

(by M. Henke)
Suggestions for team session

1. Generate a plant with parameterized leaves (parameters: length, width, ratio petiole/blade length, ...) - with interpretive rules, - with instantiation rules.

2. Create a model for a circular arrangement of mushrooms ("witches ring"). Use an instantiation rule for the multiplication and arrangement.