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

More advanced XL programming (Queries etc.)

The language XL

- extension of Java
- allows also specification of L-systems and RGGs (graph grammars) in an intuitive rule notation

procedural blocks, like in Java: { ... }

rule-oriented blocks (RGG blocks):   [...]

Features of the language XL:

- nodes of the graph are Java objects (including geometry objects)

example: XL programme for the Koch curve (see part 1)

```
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);
  ]
```

nodes of the graph

edges (type „successor“)

special nodes:

geometry objects

Box, Sphere, Cylinder, Cone, Frustum, Parallelogram...

access to attributes by parameter list:

`Box(x, y, z)` (length, width, height)

or with special functions:

`Box(...).(setColor(0x007700))` (colour) special nodes:

- geometry objects
  - Box, Sphere, Cylinder, Cone, Frustum, Parallelogram...
- transformation nodes
  - **`Translate(x, y, z)`**, **`Scale(cx, cy, cz)`**, **`Scale(c)`**, **`Rotate(a, b, c)`**, **`RU(a)`**, **`RL(a)`**, **`RH(a)`**, **`RV(c)`**, **`RG`**, **...**
- light sources
  - PointLight, DirectionalLight, SpotLight, AmbientLight

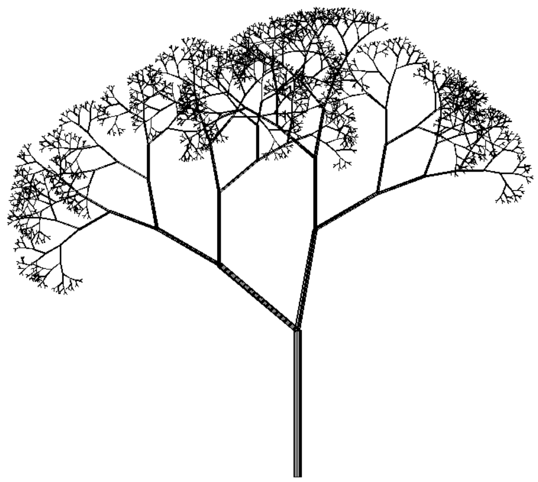
Features of the language XL:

- rules organized in blocks [...], control of application by control structures

example: rules for the stochastic tree

```
Axiom ==> L(100) D(5) A;

A ==> F0 LMul(0.7) DMul(0.7)
  if (probability(0.5))
   ( [ RU(50) A ] [ RU(-10) A ] )
  else
   ( [ RU(-50) A ] [ RU(10) A ] );
```



Features of the language XL:

- parallel application of the rules

  (can be modified: sequential mode can be switched on, see below)

Features of the language XL:

- parallel execution of assignments possible (*deferred assignment*)

special assignment operator := besides the normal =

quasi-parallel assignment to the variables x and y:

```
x := f(x, y);
y := g(x, y);
```

Features of the language XL:

- Execution rules
  - Used to execute imperative statements for searched organs or update their attribute values, without changes in the (graph) structure
  - XL syntax: `::>`

```
ModuleName ::> { imperative code; }
```

Example:

```
l:Leaf ::> { l.photosynthesis(); }
```

L-system rule

```
Internode(length, radius) ==>
    Internode(length+0.1, radius+0.01)
;
```

Execution rule

```
i:Internode ::> {
    i[length] += 0.1;
    i[radius] += 0.01;
}
```

execution rule

`A ::> {` *imperative code* **`};`**

Test the examples **`sm09_e25.rgg`**, **`sm09_e16.rgg`**, **`sm09_e17.gsz`**, **`sm09_e18.rgg`**

and concerning the access to node attributes: **`sm09_e26.rgg`**

Features of the language XL:

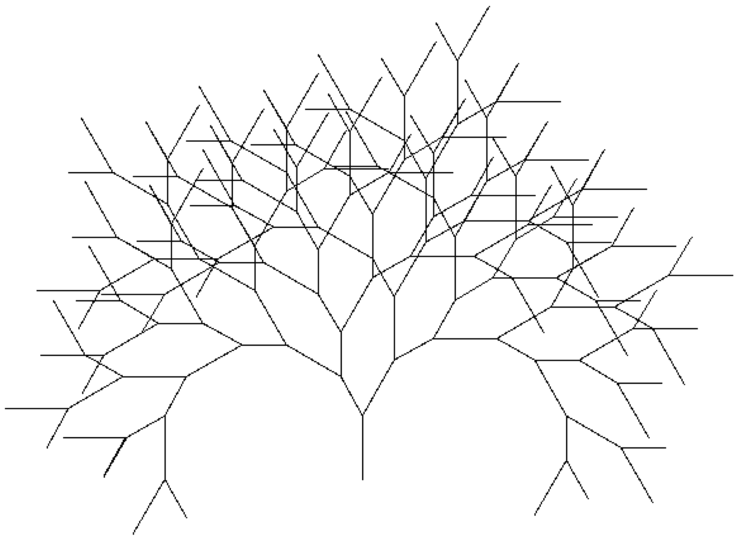
- set-valued expressions (more precisely: producer instead of sets)
- **graph queries** to analyze the actual structure

example for a graph query:

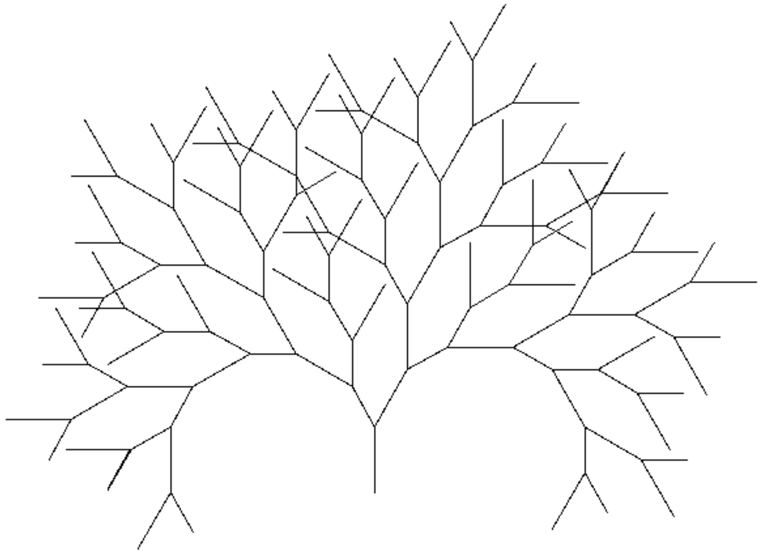
binary tree, growth shall start only if there is enough distance to other **F** objects

```
Axiom ==> F(100) [ RU(-30) A(70) ] RU(30) A(100);
a:A(s) ==> if ( forall(distance(a, (* F *)) > 60) )
           ( RH(180) F(s) [ RU(-30) A(70) ] RU(30) A(100) )
```

without the „if“ condition



with the „if“ condition



query syntax:

a query is enclosed by **`(*        *)`**

The elements are given in their expected order, e.g.:
`(* A A B *)` searches for a subgraph which consists of a sequence of nodes of the types **`A A B`**, connected by successor edges.

Queries as generalized contexts:

test the examples **`sm09_e28.rgg`**, **`sm09_e29.rgg`**, **`sm09_e30.rgg`**

Examples

- Find all internodes, and print them out

```
println((* Internode *));
```

- Find all newly created internodes (with age 0)

```
(* i:Internode, (i[age] == 0) *)
```

- Search for all internodes with diameter > 0.01

```
(* i:Internode, (i[diameter] > 0.01) *)
```

- Find all pairs with distance < 1

```
(* f:F, g:F, ((f != g) && (distance(f, g) < 1)) *)
```

- Find all nodes B, connected to A with a branching edge

```
(* A +> B *)
```

Features of the language XL:

- aggregating operators (e.g., „sum“, „mean“, „empty“, „forall“, „selectWhereMin“)

can be applied to set-valued results of a query

Queries and aggregating operators

provide possibilities to connect structure and function

example: search for all leaves which are successors of node **c** and sum up their surface areas

query

```
sum((* c (>)+ Leaf *).area)
```

transitive closure

result can be transferred to an imperative calculation

aggregation operator

Aggregate methods

- Collect multiple values and return one single value as result
- Standard aggregate operations:
  `count`, **`sum`**, **`empty`**, **`exist`**, **`forall`**, **`first`**, **`last`**, **`max`**, **`min`**, **`mean`**, **`selectRandomly`**, **`selectWhereMin`**, **`selectWhereMax`**, ...
- Can be applied to set-valued results of a query

```
count((* Leaf *))
first((* l:Leaf, (l[order] == 1 && l[rank] == 1) *))
selectRandomly((* F *))
selectWhereMax((* f:F *), (f[diameter]))
```

Examples

- Count all segments F, longer than 1

```
count((* f:F, (f[length] > 1) *))
```

- Search for all leaves and sum up their surface areas

```
sum((* Leaf *)[area])
sum((* Leaf *).area)     // alternative
```

- Sum up potentional growth rate of all growing leaves

```
sum((* l:Leaf, (l.isGrowing()) *).pgr())
```

Queries in XL

test the examples

sm09_e31.rgg,
sm09_e35.rgg,
sm09_e36.rgg

for light interception / photosynthesis:

a simple model of overshadowing

using a query referring to a geometric region in space

model approach (strongly simplifying):
overshadowing of an object occurs when there are further objects in an imagined cone with its apex in the object, opened into z direction (to the sky).

example:

`sm09_e42.rgg` competition of three 2-dimensional model plants for light

```
module Segment(int t, int ord) extends F0;
module TBud(int t) extends F(1, 1, 1);
module LBud extends F(0.5, 0.5, 1);

Vector3d z = new Vector3d(0, 0, 1);

protected void init()
   [
   Axiom ==> P(2) D(5) V(-0.15) [ TBud(-4) ] RU(90) M(600) RU(-90)
                                [ TBud(0) ] RU(-90) M(1200) RU(90)
                                [ TBud(-8) ];
   ]

public void run()
   [
   TBud(t), (t < 0) ==> TBud(t+1);
   x:TBud(t), (t >= 0 && empty( (* s:Segment, (s in cone(x, z, 45)) *) ) ) ==>
      L(random(80, 120)) Segment(0, 0)
      [ MRel(random(0.5, 0.9)) RU(60) LBud ]
      [ MRel(random(0.5, 0.9)) RU(-60) LBud ] TBud(t+1);
   y:LBud,
      (empty( (* s:Segment, (s in cone(y, z, 45)) *) ) ) ==>
      L(random(60, 90) Segment(0, 1) RV0 LBud;
   Segment(t, o), (t < 8) ==> Segment(t+1, o);
   Segment(t, o), (t >= 8 && o == 1) ==>> ;   /* removal of the whole branch */
   ]
```

Representation of graphs in XL

- node types must be declared with „**`module`**“

- nodes can be all Java objects.
  In user-made **`module`** declarations, methods (functions) and additional variables can be introduced, like in Java

- notation for nodes in a graph:
  `Node_type`, optionally preceded by: **`label:`**
  Examples: **`A`**, **`Meristem(t)`**, **`b:Bud`**

- notation for edges in a graph:

  `-`*edgetype***`->`**, **`<-`***edgetype***`-`**

- 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)

user-defined edge types

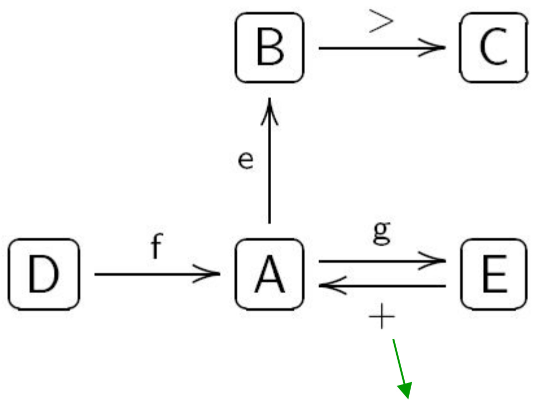
`const int xxx = EDGE_0;` **`//`** oder **`EDGE_1`**, ..., **`EDGE_14`**

...

usage in the graph: **`-xxx->`**, **`<-xxx-`**, **`-xxx-`**

Notation of graphs in XL

example:



(>: successor edge, +: branch edge)

is represented in programme code as

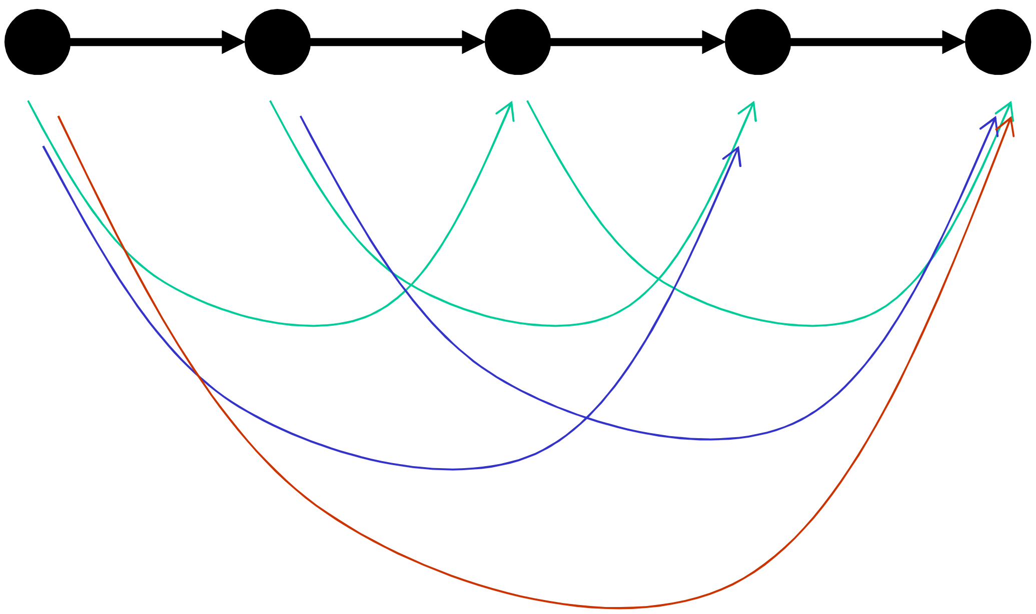
```
a:A [-e-> B C] [<-f- D] -g-> E [a]
```

(the representation is not unique!)

Derived relations

Relations between nodes connected by more than 1 edge

E.g.: Relation between nodes connected by several edges (one after the other) of the same type:



„transitive closure“ of the original relation (edge)

XL notation

- 1-to-*n* repetitions: **+**

```
A (-edgetype->)+ B
```

- 0-to-*n* repetitions: $\star$

```
A (-edgetype->)* B
```

- *Minimal elements* (stop searching once a match has been found):

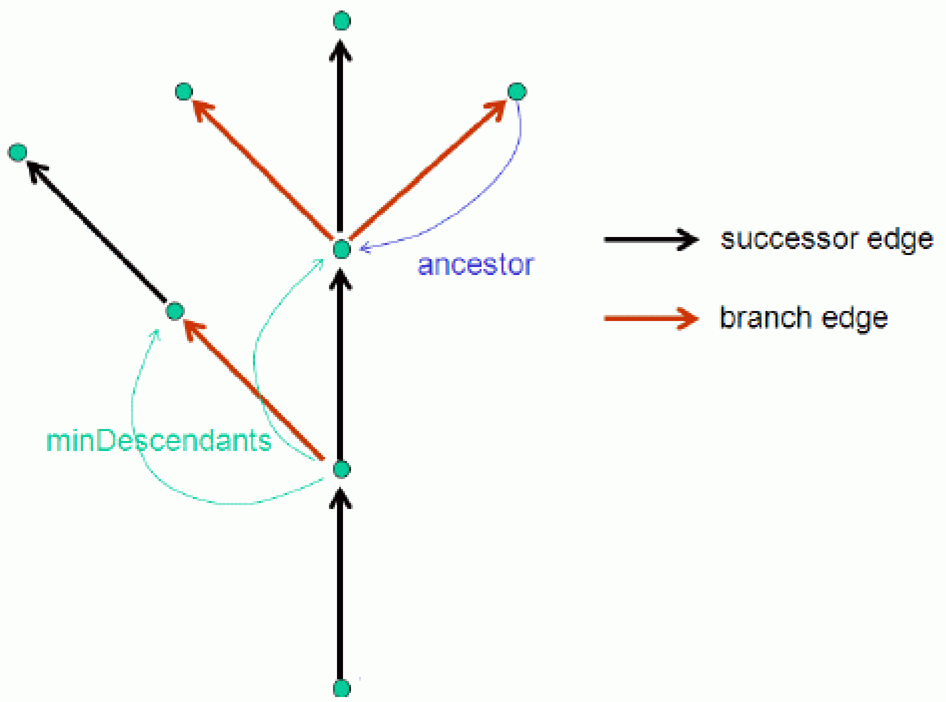
```
A (-edgetype->)+ :(B)
A (-edgetype->)* :(B)
```

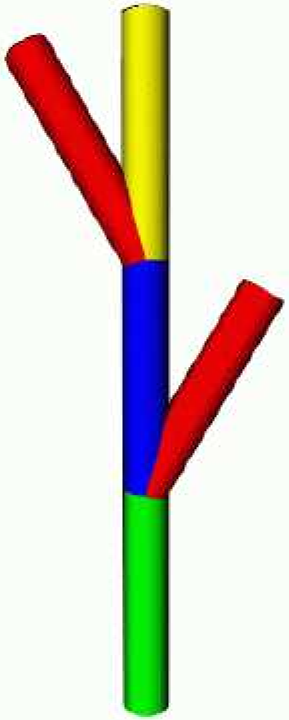
Examples

- Find all/the first internode connected to the leaf

```
(* Leaf (<--)+ Internode *)
(* Leaf (<--)+ :(Internode) *)
```

ancestor - nearest preceding node of a certain node type
minDescendants - nearest successors of a certain node type (nodes of other types are skipped)
descendants - all successors of a certain node type





```
// all modules extend the module Internode
Axiom ==>
    InternodeFirst [RU(30) BranchFirst]
    InternodeSecond [RU(-30) BranchSecond]
    InternodeThird
;
// queries and their outputs:
(* InternodeFirst -minDescendants-> Internode *)
    BranchFirst
    InternodeSecond

(* InternodeFirst -descendants-> Internode *)
    InternodeSecond
    InternodeThird
    BranchSecond
    BranchFirst

(* InternodeThird -ancestor-> Internode *)
    InternodeSecond

(* InternodeThird (-ancestor->)+ Internode *)
    InternodeSecond
    InternodeFirst
```

The current graph

GroIMP maintains always a graph which contains the complete current structural information. This graph is transformed by application of the rules.

Attention: Not all nodes are visible objects in the 3-D view of the structure!

- **`F0`**, **`F(x)`**, **`Box`**, **`Sphere`**: yes

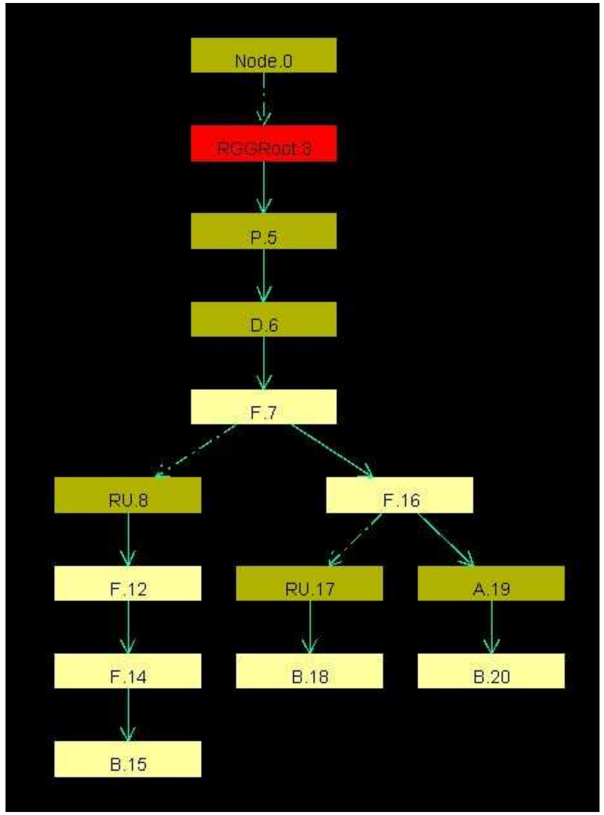
- **`RU(30)`**, **`A`**, **`B`**: normally not (if not derived by „**`extends`**“ from visible objects)

The graph can be completely visualized in the **2-D graph view** (in GroIMP: Panels - 2D - Graph).

Load an example RGG file in GroIMP and execute some steps (do not work with a too complex structure).

Open the 2-D graph view, fix the window with the mouse in the GroIMP user interface and test different layouts (Layout - Edit).

Keep track of the changes of the graph when you apply the rules (click on „redraw“)!



which parts of the current graph of GroIMP are visible (in the 3-d view) ?

all geometry nodes which can be accessed from the root (denoted ^) of the graph by exactly one path, which consists only of "successor" and "branch" edges

How to enforce that an object is visible in any case:

```
==>> ^ Object
```