Relational Growth Grammars and the Programming Language XL

Ole Kniemeyer

July 26, 2007

Outline

Motivation behind Relational Growth Grammars

- L-systems are quite successful, but based on plain strings.
- Modern representation of 3D worlds: scene graphs

Combine L-systems and graphs: parallel graph grammars

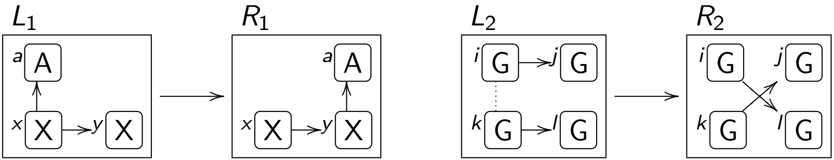
- Intensively studied in the mid and late 1970s
- Fell into desuetude
- L-systems still in active use
- Revival of parallel graph grammars needed!

Relational growth grammars (RGG) are an attempt for such a revival.

Graph Grammars

- Operate on graphs.
- Productions (rules): $L \rightarrow R$ with graphs L, R
- Some embedding specification: how to connect R with rest?

Embedding of gluing type: identification of parts of L and R



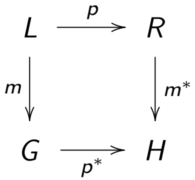
- Relatively easy to understand.
- Not suitable if R makes no reference to L (L-system rules!)

Graph Grammars

Productions with gluing: pure algebraic theory possible

- Production $L \longrightarrow R$
- Identification $p : L \longrightarrow R$
- Current graph G
- Match $m : L \longrightarrow G$

$\Rightarrow$ derived graph H as pushout



Allows quite elegant and abstract constructions and proofs.

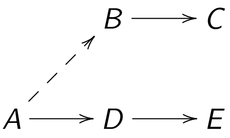
Parallelism

How to apply productions in parallel?

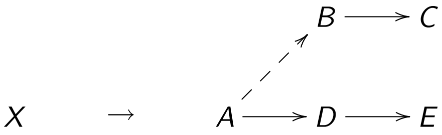
- Given pairs (p_i, m_i) of productions $p_i : L_i \rightarrow R_i$ and matches $m_i : L_i \rightarrow G$
- Parallel derivation is well defined by production $\sum_i p_i$ and match $\sum_i m_i$

L-system strings with brackets as graphs

Represent strings directly as graphs! $A[BC]DE$ becomes to



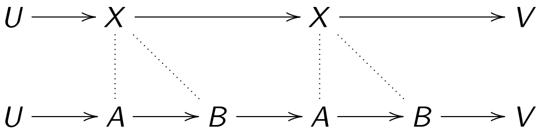
L-system production $X \rightarrow A[BC]DE$ becomes to



Problem: no identification between L and R

Connection transformation as embedding

Connection transformations establish desired edges:
$X \rightarrow AB$



- Move incoming edges from X to A.
- Move outgoing edges from X to B.

Programming language XL as extension of Java

XL: complete extension of Java

Design guidelines: generality and smooth integration into Java

Generality:

- not only suitable for a fixed graph model
- not only suitable for relational growth grammars

Smooth integration into Java:

- consistent syntax, following the C tradition
- possibility to mix old Java and new XL code

Rules within XL and a unified view thereof

Three diffent kinds of rules are defined:

- structural L-system-like rule

```
Bud ==> Internode [RU(40) Bud] Bud;
```

- structural graph rule with gluing

```
Gene i, j, k, l;
i j, k l, i -aligned- k ==>> i l, k j;
```

- execution rule

```
x:Tree ::> x.age++;
```

Right-hand sides can be viewed as sequences of statements.
Common structure of rules:

- left-hand side: query in current structure
- right-hand side: statements

Special syntax for right-hand sides of structural rules!

Structure queries find occurences of patterns

`F(len)`	find all nodes of class `F`, bind value of its length to local variable `len`
`x:Cell [a:Ant] y:Cell`	find all consecutive `Cell`s such that an `Ant` sits on the first `Cell`
`a:Ant -sees-> b:Ant`	find all pairs of `Ant`s such that the first sees the seconds

- textual notation
- syntax resembles syntax of L-systems
- composed of predicates
  - `Cell`: type predicate
  - `F(len)`: module predicate
  - `-sees->`: relational predicate implemented by boolean method

Structure is defined by data model

- Queries access structure through data model interface.
- Interface can be implemented for every graph-like structure: real graphs, trees, XML documents, ...
- Ensures generality at the level of queries.

Right-hand sides are defined by operator overloading

Usual L-system syntax as ideal: `A(x) ==> F(x) A(x*0.5)`

Possible approximation with C++-style operator overloading:
`producer << new F(x) << new A(x*0.5);`
where `producer` is responsible for structure creation.

XL: special translation scheme for right-hand sides:

1. Implicit creation of new node instances:
   `producer ␣ new F(x) ␣ new A(x*0.5)`
2. “Space operator” is mapped to method `operator$space`

```
producer.operator$space(new F(x))
          .operator$space(new A(x*0.5))
```

Further example for use of operators

```
x:X ==>> x > A -branch-> B;
```

1. Implicit creation of new node instances:
   ```
   producer ␣ x > new A() -> (new B(), branch)
   ```
2. Mapping to method names
   ```
   producer.operator$space(x)
           .operator$gt(new A())
           .operator$arrow(new B(),branch)
   ```

- Definition without any reference to data model
- Ensures generality at the level of right-hand sides.

How to implement parallelism?

Conflict for the implementation of parallelism:

- Computers (Java Virtual Machines) work sequentially.
- Graph has to remain constant during derivation.

Two solutions:

1. Derivation creates completely new graph (traditional solution of L-system software).
2. Enqueue intended modifications, collectively apply them at end of derivation to graph.

Best solution: several queues

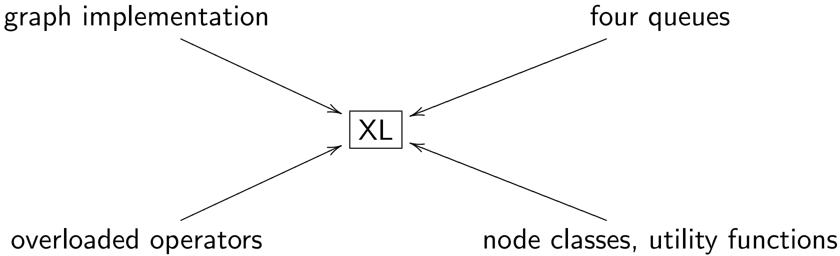
Queue-based solution is more efficient.
Corresponds to application of parallel production = sum of component productions.

Four queues for RGG:

1. Addition of connection edges
2. Addition of nodes, edges
3. Deletion of nodes, edges, dangling edges
4. Modification of properties

Order solves deleting/preserving conflicts in favour of deletion.

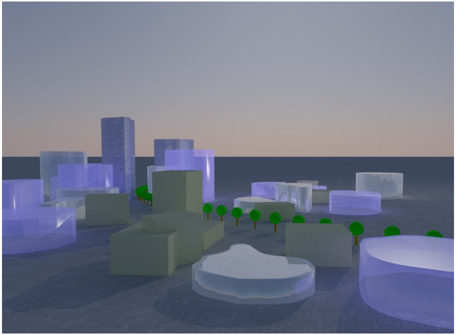
Relational growth grammars implemented by XL




```
Bud ==> Internode [RU(40) Bud] Bud;
```

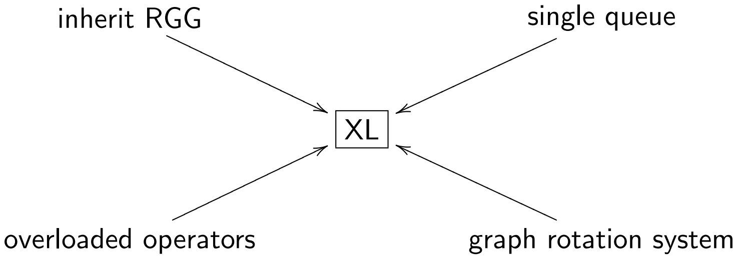
Applications

Barley model



City generator

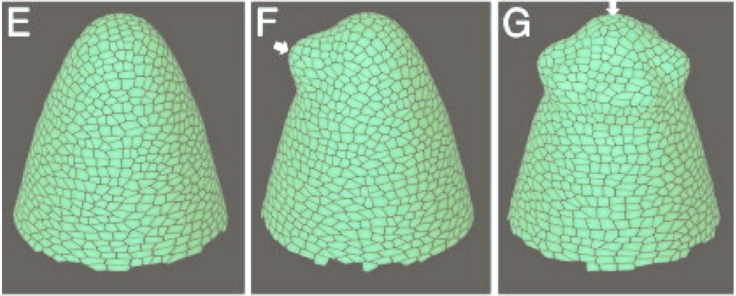
Vertex-vertex algebras implemented by XL




```
... v [p r q], r v in p, q v in r, p v in q;
```

- No need for copy of "old graph".
- Rule-based instead of imperative formulation.

Potential application



Shoot apical meristem:
application of auxin, development of primordia

(from Smith et al.: A plausible model of phyllotaxis (2006))

Conclusion

Relational growth grammars:

- Parallel graph grammars
- L-systems as special case

XL:

- Programming language extending Java
- Queries find matches for patterns in structure, structure defined by data model interface
- Operator overloading

RGG implemented on the basis of XL:

- Implementation of data model interface for graphs
- Implementation of operator overloading methods
- Usage of queues to achieve parallelism

Outlook

- Three-dimensional meshes (volumes)
- Interfacing with differential equations
- Usage of graph scheme to check consistency

The End

Thank you for your attention!