Rule-based modelling with the XL/GroIMP software

Ole Kniemeyer
Brandenburgische Technische Universität Cottbus
Institut für Informatik
Lehrstuhl Praktische Informatik / Graphische Systeme
P.O. Box 101344
D-03013 Cottbus, Germany
okn@informatik.tu-cottbus.de

Abstract

In this software demonstration, the concept of "Relational Growth Grammars" will be illustrated in its concrete implementation, namely the high-level language XL which combines the rule-based programming paradigm of graph grammars and L-systems with the imperative and object-oriented programming paradigm of Java. The suitability of XL as a description language of rule-based ALife models will be shown in several examples.

1 Introduction

There exist numerous ALife models whose specification is done in a rule-based manner, the most famous of these probably being Conway's Game Of Life. However, the implementation of these models is mostly carried out with conventional imperative programming languages like C or Java, where one has to emulate the rule-based specification using additional code of technical nature. Such a technical overhead reduces the simplicity and clarity of the implementation.

In the field of plant modelling, the rule-based programming approach of L(indenmayer)-systems [11] has been widely and successfully used to model growth and architecture of individuals. L-system-based programming languages allow a model implementation which is highly congruent with its specification, but their field of application is restricted due to the constraints inherent in this kind of string rewriting grammars.

Several extensions to the L-system formalism have been made, among them "globally sensitive" and "open" L-systems which take the environment into account [9, 10]. However, all these extensions remain within the framework of string rewriting grammars. A much wider range of application can be achieved by the transition from string to graph grammars. In the concept of "Relational Growth Grammars" [7], this step has been done, resulting in the new programming language XL which combines L-systems, graph grammars and Java. In conjunction with the interactive modelling platform GroIMP, complex models of biology, but also of ALife, can be implemented in a flexible way which is close to the requirements of the system in question.

In this article, the language XL, the software platform GroIMP and the underlying "Relational Growth Grammars" will be introduced informally, using simple but paradigmatic examples.

2 Relational Growth Grammars

The concept of Relational Growth Grammars (RGG) has been (and still is being) developed within a project in the framework of plant modelling. The aim is the definition and implementation of a suitable rule-based language to be used for the modelling of architecture, processes and interactions during plant growth. Though this immediate field of application is concerned with the description of botanical real life, ALife models form a helpful basis for the abstraction of the requirements and, ultimatively, the design of such a language.

The "historic roots" of RGG are founded in L-systems and their extension to sensitive growth grammars [9]. L-systems are a variant of string rewriting grammars, applied in parallel. Combined with a turtle graphics interpretation of string symbols, realistic images of, e.g., plant morphology and development can be produced. A famous and simple example is the von Koch (snowflake) curve [8], which can be constructed out of a start word α by repeated application of the rule system

$$\begin{aligned} \alpha &\longrightarrow \texttt{F + + F + + F} \\ \texttt{F} &\longrightarrow \texttt{F - F + + F - F} \end{aligned}$$

`F`, `+` and `-` are commands for the turtle and mean *move and draw a line*, *turn left by 60°* and *turn right by 60°* respectively.

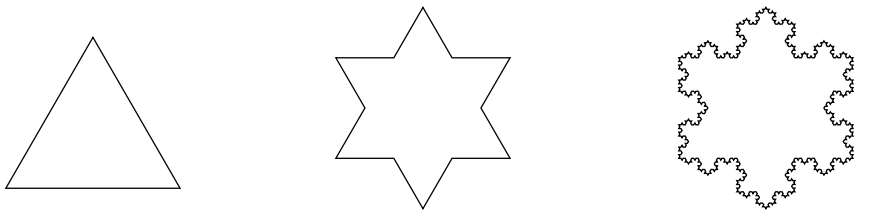


Figure 1: The first steps of the snowflake L-system.

The rule-based approach in itself is well suited for the specification of biological or ALife models, but in its concrete form of L-systems, the restriction to strings as fundamental objects and string grammars as dynamics narrows the field of application:

- Structural (morphological) aspects are easily implemented, but functional (process oriented) aspects lack a coherent representation.
- The plain string nature of objects prevents the use of object oriented techniques.
- The operation of complex interactions on the string-encoded structures is neither efficient nor reasonably realizable.

At present, solutions for these problems are mostly going in the direction of "outsourcing" instead of "integration": Outside the L-system itself, the L-system interpreter provides a set of predefined interaction functions [9], or provides an interface to external programmes [10].

RGGs overcome these problems in an integrative way:

- The requirements of functional aspects are addressed by the inclusion of a "conventional" object oriented programming language (this has been done in [5] with C++).

- The requirements of complex structures and interactions – global or local – are addressed by the transition from strings to graphs, from string grammars to graph grammars [12], thereby retaining the structural view of the L-system paradigm.
- To fully integrate the fundamental graph structure, a graph query language is included.

RGG graphs are edge-labelled directed graphs, in which the nodes are objects of the underlying programming language. Edges form a simple, static kind of relation between the incident nodes, e.g., a distinct edge type may denote the *successor*-relation in a string-like structure. In this way, other common data structures like sets, multisets or trees can be represented.

In RGGs, the definition of more complex relations between two nodes is possible. A relation could be the membership of node a to the neighbourhood of node b, possibly defined geometrically and not in terms of edges, or the shading of a plant leaf a by another leaf b, possibly dependent on the position of the sun. This relational view on objects is used in graph queries or the left hand (pattern) side of rules.

3 XL: A Java implementation of Relational Growth Grammars

The programming language XL is a concrete implementation of Relational Growth Grammars. It is designed as a language for the use in practice, e.g., plant modelling, and not just as a proof of concept. Java has been chosen as the underlying programming language, and Java served as a basis for the language XL itself. Currently, most of the constructs of the Java language [4] have been integrated. For instance, the classical "Hello World" programme, written in XL, is identical to the Java "Hello World" programme. The entire range of existing Java runtime libraries can be used.

More interesting than "Hello World" is, of course, the translation of the snowflake L-system into XL:

```
class Koch extends RGGSystem
{
    void run()
    [
        Axiom ==> F(1) RU(120) F(1) RU(120) F(1);
        F(x) ==> F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3);
    ]
}
```

The `+`- and `-`-rotations have been replaced by `RU`-rotation nodes. The additional length parameter of `F`, which is scaled by $\frac{1}{3}$ in each step, ensures that the total size of the figure remains the same.

In this example, `Axiom`, `F` and `RU` correspond to Java classes of that name. The actual graph consists of instances (nodes) of these classes, which are – in this simple example – linearly connected by edges of type *successor*. The application of the rule system `run` to a given graph is done as in L-systems; e.g., if an instance of class `Axiom` is found in the graph, it is replaced by a sequence of new instances of `F`- and `RU`-nodes.

The necessary embedding Java-like code makes the XL snowflake code longer than its pure L-system variant. In this simple example, this may seem disadvantageous, but when looking at more complex examples, the possibility of grouping rules and giving them a name helps structuring the code – a well known fact in programming.

In the snowflake example, nothing is gained by the transition from L-systems to XL: The fractal snowflake construction ideally lies in the very domain of L-systems. This situation changes in the next example, Conway's Game Of Life. It is defined as a cellular automaton on a rectangular grid, each grid cell having two states: Dead or alive. The state transition is composed of two rules:

- If a living cell has less than two, or more than three living neighbours (the eight surrounding cells), the cell dies.
- If a dead cell has exactly three living neighbours, it becomes alive.

Whilst the implementation of this cellular automaton would not be natural in the framework of L-systems, an XL implementation is straightforward. Using the Java class `Cell`, which has an associated parameter field `state`, the definition of neighbourhood and transition rules can be done as follows:

```
iterating Cell neighbours(Cell c1)
{
    yield (* c1 -+- #c2:Cell, (c1.distanceLinf (c2) < 1.1) *);
}

void transition()
[
    x:Cell(1), (!(sum(neighbours(x).state) in {2..3})) ==>> x(0);

    x:Cell(0), (sum(neighbours(x).state) == 3) ==>> x(1);
]
```

The method `neighbours` returns multiple values (indicated by the keyword `iterating`) through the `yield`-statement, namely all those cells `c2` which are connected by an arbitrary path (`-+-`) to `c1` and lie within an L_∞-radius 1.1 (i.e., within an axis-parallel square of half side length 1.1) around `c1`. Assuming a grid distance of 1, this is a geometrical definition of the Game Of Life-neighbourhood.

In the rules of the method `transition`, the sum over the states of the `neighbours` of a cell instance `x` is built and used in the rule application conditions. If, e.g., `x` is in state 0 and the sum equals 3, `x` replaces itself, i.e., no new node is created, no structural change to the graph is done, and the state of `x` is set to 1, in accordance with Conway's definition.

This Game Of Life implementation demonstrates three new features of XL:

- Inside starred parentheses, graph queries can be formulated which search in the graph for a given pattern and return all matches successively.
- Expressions having multiple values successively are possible: Graph queries or the range operator {`a..b`} are such expressions, methods having multiple values can be defined using `iterating` / `yield` (similar constructs are known in other programming languages like CLU, Python, Ruby). Operators like `sum` or `in` use these expressions as operands.
- Node patterns of the left hand side of a rule can be labelled (`x:`), these labels can be used on the right hand side to specify that and where the matching nodes shall remain in the graph.

By defining the neighbourhood as a relation, another implementation is possible:

```
boolean neighbour(Cell c1, Cell c2)
{
    return c1.distanceLinf(c2) < 1.1;
}

void transition()
[
```

```
    x:Cell(1), (!(sum((* x -neighbour-> #Cell *).state) in {2..3})) ==>> x(0);

    x:Cell(0), (sum((* x -neighbour-> #Cell *).state) == 3) ==>> x(1);
]
```

Besides the surrounding lines, the Game Of Life dynamics is implemented in essentially three lines of code. These are highly congruent with the specification by Conway: The definition of neighbourhood, the rule for living cells and the rule for dead cells. An important point is the application of rules in parallel: Cellular automatons work this way, but also processes occurring in nature are best modelled using parallel application.

4 The modelling platform GroIMP

The modelling platform GroIMP is designed as an integrated platform which incorporates modelling, visualisation und interaction. It exhibits several features which makes itself suitable for the field of biological or ALife modelling:

- The "modelling backbone" consists in the language XL. It is fully integrated, e.g., the user can interactively select the rules to be applied.
- GroIMP provides classes that can be used in modelling: Turtle commands, further geometrical classes like bicubic surfaces, the class `Cell` which has been used in the Game Of Life implementation, and so on.
- The outcome of a model is visualised within GroIMP.
- In the visual representation of the model output, users can interact with the dynamics of the model, e.g., by selecting or deleting elements. A networked mode is available, allowing different users to interact with the modelled world synchronously. This may be an interesting feature to be used in the field of e-learning.
- The networked mode may also be useful when modelling complex systems, as it allows for grid computing. The rules of L-systems or relational growth grammars are to be applied in parallel in any case, so the use of a grid instead of a single computer is a natural choice, and the modeller can benefit from this powerful technique without any change in the model code. This networked mode is currently in an experimental stage.

Fig. 2 shows the visualisation of the previous Game Of Life example within GroIMP. Using the mouse, a user could click on a cell to change its state manually.

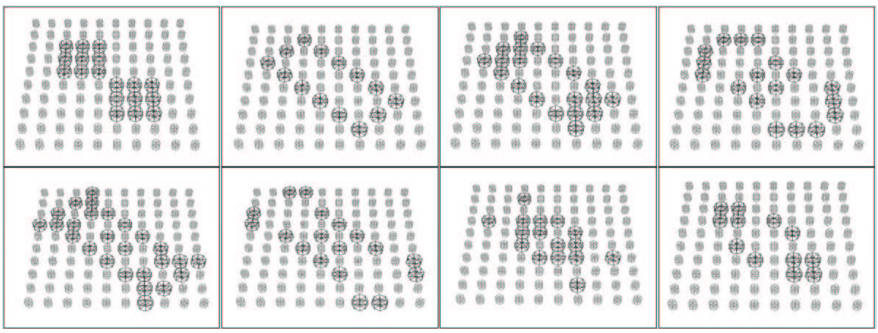


Figure 2: GroIMP visualisation of the Game Of Life: A period-8-oscillator.

5 More examples

5.1 A simple ant model

Ants are a popular subject in ALife. Real ants communicate with each other by pheromone trails laid down during movement. An ant encountering an existing trail follows this trail with high probability and, in doing so, intensifies the trail. This positive feedback loop leads to a remarkable collective behaviour of an ant population: The shortest paths between food sources and the nest are found and then followed very quickly [3].

A simplistic simulation can be implemented easily using XL. The ant agents move in a rectangular grid and have the following properties:

- An "excitation state" determines the amount of pheromone to be laid down at the current grid cell. Its value is increased when the ant is on a food source cell, otherwise it is decreased in each time step.
- A memory records the last twenty cells visited.
- In each time step, the cell to move to is chosen among the eight neighbouring cells which have not yet been recorded in the memory. The choice is influenced by the pheromone values of the cells, a tendency to keep the current direction and a random effect.

The rectangular grid consists of cells containing an amount of pheromone which decays step by step. Cells may be marked interactively by the user as food sources.

In the following implementation, the excitation state of an ant and the pheromone content of a cell are stored in the `length`-variable of the representing cylinders. This use has the advantage of a direct visualisation of the corresponding values.

```
class Ant extends Cylinder {
    float dx, dy; // current moving direction
}

class AntSimulation extends RGGSystem {
    const int memory = MIN_USER_EDGE; // a user-defined edge

    boolean placeAntAt(int i, int j) { where to place ants initially... }

    float evaluate(float pheromone, float dirdelta2) { evaluation of a possible move...
        // pheromone: Pheromone content of the reached cell,
        // dirdelta2: squared difference of the direction vectors }

    Cell nextCell(Cell c1, Ant a) {
        float dx, dy;
        // find the neighbouring cell c2 not in memory with maximum evaluate-value:
        Cell next = select((* c1 -+- #c2:Cell,
                            (c1.distanceLinf (c2) < 1.1),
                            (!exist((* a -memory-> int c2 *))) *),
                           (dx = (c2.x - c1.x) - a.dx, dy = (c2.y - c1.y) - a.dy,
                            evaluate(c2.length, dx * dx + dy * dy)) -> max);
        return (next != null) ? next
                              : c1; // all neighbours are in memory, ant doesn't move
    }

    void init() [
        // create a 25 * 25 grid of cells, place the ants
        Axiom ==>> ^ for(i = 0 .. 24) for(j = 0 .. 24) ([Cell(i, j) if(placeAntAt(i, j)) Ant]);
    ]
```

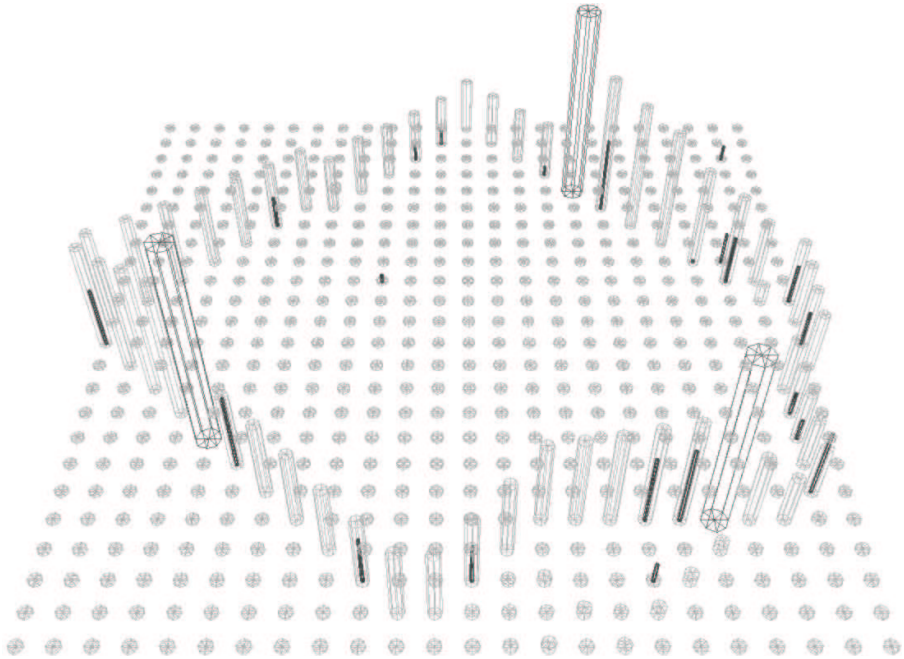


Figure 3: Snapshot of the ant model: Ants are represented as thin, black cylinders; they have found a round trip which passes the three food sources.

```
    void run() [
        // move ant a to next cell, keep current cell c in memory with counter value 20
        c:Cell a:Ant ==>> n:nextCell(c, a) a -memory-> 20 c
                   { a.dx := n.x - c.x; a.dy := n.y - c.y; // update moving direction
                     float p = (a.length + c.state) * C_ANT; // amount of pheromone laid down
                     a.length :+= c.state - p; // update ant excitation status
                     c.length :+= p; // lay down pheromone };

        // decrease the memory counter; if it has reached zero, the memory node is removed
        m:int ==>> if (m > 0) m(m-1);

        c:Cell ::> c.length :-= c.length * C_CELL; // decay of cell pheromone
    ]
}
```

The colon-prefixed assignment operators `:=`, `:+=` and `:-=` aren't executed immediately: As for structural changes to the graph, these modifications are written to a modification protocol and executed at once after all rules have been processed. This ensures the parallelity of rule application.

While running this example, food sources may be set or removed, whole cells may even be removed in order to simulate obstacles. After a while and provided that the model is suitably parameterized (`C_CELL`, `C_ANT`, `evaluate`), the ants will find short trails connecting the available food sources. These pheromone trails can be seen in Fig. 3.

5.2 ABC model of flower morphogenesis

As seen from the perspective of plant modelling, the previous examples are a slight digression into non-botanical ALife. In particular, the power of L-systems has not been used. Returning to botany, this aspect of RGG immediately becomes essential, but now, within the framework of RGG, the expressiveness of L-systems is considerably enhanced.

A good example of the new possibilities is flower morphogenesis which is controlled by a genetic regulatory network. The ABC model of flower morphogenesis describes this process using three genes A, B and C, with corresponding transcription factors [2]. According to the dynamics of a regulatory network, these factor concentrations change in time, thereby leading to a time-variant expression of functions. Depending on the current state of expression, different flower organs are formed. Finally, the sequence and type of organs formed constitutes the mature flower.

In [6], the formal language `L-transsys` is presented which addresses the requirements of such an integrated model of flower morphogenesis. However, both model levels, the L-system controlling morphogenesis and the regulatory network, reside in their own, separate domains of this language. Using RGG, a further unification can be made.

The XL implementation of the ABC model is a direct translation of the implementation of [6]. The Michaelis-Menten kinetics of the regulatory network used there is translated into the rules

```
f:Factor(c, d) ::> f.concentration :-= c * d;

f:Factor <-encodes- g:Gene(ct) ::> f.concentration :+= Math.max (0,
        sum(((* Factor(c2,) Activate(s, m) g *), m * c2 / (s + c2))) + ct);
```

Here, the classes `Factor` with associated parameters *concentration* and *decay*, `Gene` with associated parameter *constitutive* and `Activate` with the associated Michaelis-Menten parameters *specificity* and *maximal rate* are used. The details can be found in [6].

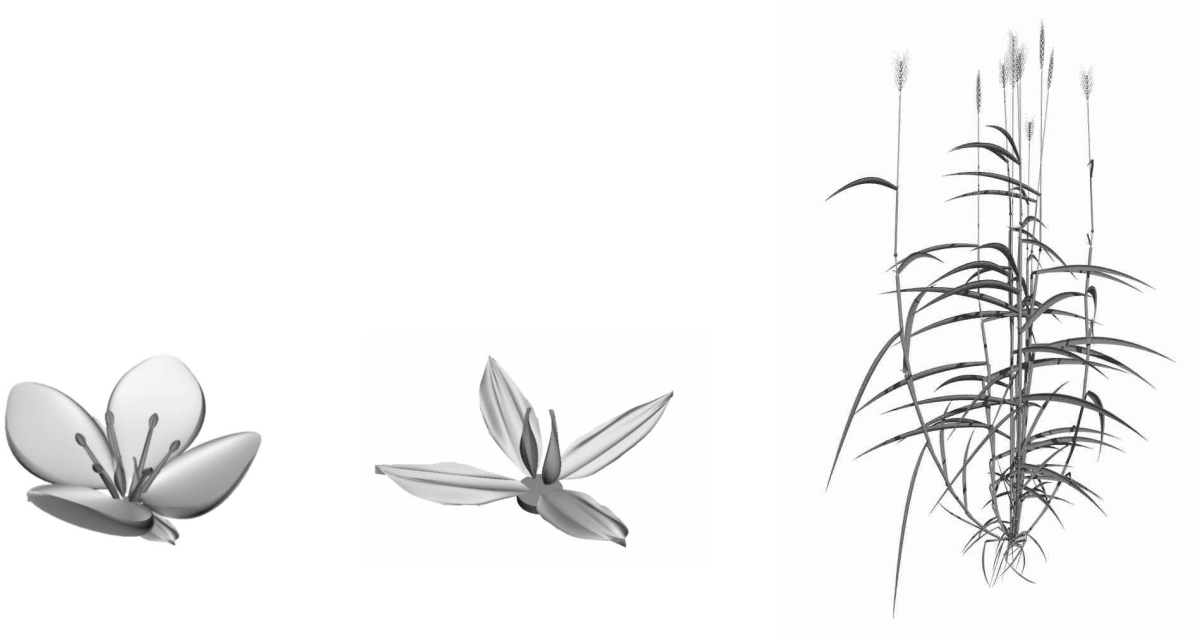
The ABC network proper is constructed in the initialization rule using code like

```
...
agene:Gene(0.1) -encodes-> a:Factor(0, 0.3),
a Activate(1e-9, 50) agene,
...
```

The flower meristem is now connected to this network and uses the current factor concentrations to determine the new organ to form:

```
m:Meristem (* -factors-> Factor(a,) Factor(b,) Factor(c,) *) ==>
    {   // determination of next organ type
        int t = (b > 80) ? ((c > a) ? STAMEN : PETAL)
              : (a > 80) ? ((c > 80) ? SHOOT : SEPAL)
              : (c > 80) ? CARPEL : PEDICEL;
    }
    if (t == SHOOT) (
        F(0.5, 0.6)
    )
    else if (t == PEDICEL) (
        ...
    )
    ...
    m;
```

This L-system style rule establishes the connection between the regulatory network and the morphogenesis – all within the framework of RGG. In Fig. 4, the graphical output of this ABC model – dressed up with some computer graphics – is shown for the wildtype and a mutant, where an edge in the regulatory network has been cut to model the loss of a function. Such mutants are observed in nature.



ABC wild type "Loss of B" mutant Barley model

Figure 4: Simulated phenotype of wild type and mutant flower. Graphical output of the barley model.

5.3 Barley morphogenesis controlled by genotype and a metabolic network

The last example is a model of barley morphogenesis. It is work in progress and will only be sketched here. The model consists of the following components:

- The morphogenesis of tillers and ear is modelled in an L-System style based on [1].
- A diploid genome controls ear morphogenesis. Genomes can be mutated or crossing-over being applied.
- A metabolic network in each internode organ controls internode elongation. Metabolites flow along blades and internodes.

Currently, the parameterization of the metabolic network is *ad hoc* and to be understood only as a proof of concept. Fig. 4 shows the outcome of the model.

6 Outlook

The examples have shown that Relational Growth Grammars are capable of representing models from different fields with different underlying mechanisms in a concise way which retains the structural view of L-systems. The necessity of technical, non-model-inherent code has been reduced as compared to conventional programming languages. Further models will be implemented in order to check the suitability of RGGs, possibly resulting in the introduction of new language features.

One very important topic, especially when dealing with large systems, is the runtime efficiency. Compared to a "direct" implementation in a conventional programming language, graph grammars introduce a runtime overhead which may become considerable in certain situations; e.g., the geometrical definition of the Game Of Life neighbourhood is inherently inefficient when the graph matching algorithm works in the naïve way (as it is currently the case). Further efforts have to be made to improve

the efficiency of the matching algorithm, e.g., an optimization of search order or a caching of matches found in previous steps. In any case, these technical problems are hidden to RGG users and models.

7 Acknowledgements

This research is joint work with Gerhard Buck-Sorlin and Winfried Kurth. It was funded by the DFG under grant Ku 847/5-1 in the framework of the research group "Virtual Crops". All support is gratefully acknowledged.

References

[1] Buck-Sorlin, G.H., Bachmann, K.: Simulating the morphology of barley spike phenotypes using genotype information. Agronomie: Plant Genetics and Breeding **20** (2000) 691–702.

[2] Coen, E.S., Meyerowitz, E.M.: The war of the whorls: genetic interactions controlling flower development. Nature **353** (1991) 31–37.

[3] Colorni, A., Dorigo, M., Maniezzo, V.: Distributed Optimization by Ant Colonies. In: Varela, F., Bourgine, P. (eds.): Proceedings of the First European Conference on Artificial Life, Paris, France. Elsevier Publishing (1992) 134–142.

[4] Joy, B., Steele, G., Gosling, J., Bracha, G.: The Java Language Specification, Second Edition. Addison-Wesley, Reading (2000)

[5] Karwowski, R., Prusinkiewicz, P.: Design and Implementation of the L+C Modeling Language. In: Giavitto, J.-L., Moreau, P.-E. (eds.): Electronic Notes in Theoretical Computer Science **86** (2) (2003)

[6] Kim, J.: `transsys`: A Generic Formalism for Modelling Regulatory Networks in Morphogenesis. In: Kelemen, J. et al. (eds.): Advances in Artificial Life. Lecture Notes in Artificial Intelligence **2159**, Springer, Berlin Heidelberg (2001) 242–251.

[7] Kniemeyer, O., Buck-Sorlin, G.H., Kurth, W.: Representation of genotype and phenotype in a coherent framework based on extended L-systems. In Banzhaf, W. et al. (eds.): Advances in Artificial Life. Lecture Notes in Artificial Intelligence **2801**, Springer, Berlin Heidelberg (2003) 625-634.

[8] Koch, H. von: Une méthode géométrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes. Acta Mathematica **30** (1906) 145–174.

[9] Kurth, W.: 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. Berichte des Forschungszentrums Waldökosysteme der Universität Göttingen, Ser. B, **38** (1994).

[10] Měch, R., Prusinkiewicz, P.: Visual models of plants interacting with their environment. ACM SIGGRAPH 1996, New Orleans, ACM (1996) 397–410.

[11] Prusinkiewicz, P., Lindenmayer, A.: The Algorithmic Beauty of Plants. Springer-Verlag, New York (1990).

[12] Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations. World Scientific, Singapore (1997).