Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

vorgelegt von

Diplom-Physiker
Ole Kniemeyer

geboren am 27. Juni 1977 in Bremen

Gutachter:	Prof. Dr. Winfried Kurth
Gutachter:	Prof. Dr. Claus Lewerentz
Gutachter:	Prof. Dr. Hans-Jörg Kreowski
Gutachter:	Prof. Dr. Dr. h.c. Branislav Sloboda

Tag der mündlichen Prüfung: 21. November 2008

Abstract

Increasing biological knowledge requires more and more elaborate methods to translate the knowledge into executable model descriptions, and increasing computational power allows to actually execute these descriptions. Such a simulation helps to validate, extend and question the knowledge.

For plant modelling, the well-established formal description language of Lindenmayer systems reaches its limits as a method to concisely represent current knowledge and to conveniently assist in current research. On one hand, it is well-suited to represent structural and geometric aspects of plant models – of which units is a plant composed, how are these connected, what is their location in 3D space –, but on the other hand, its usage to describe functional aspects – what internal processes take place in the plant structure, how does this interact with the structure – is not as convenient as desirable. This can be traced back to the underlying representation of structure as a linear chain of units, while the intrinsic nature of the structure is a tree or even a graph. Therefore, we propose to use graphs and graph grammars as a basis for plant modelling which combines structural and functional aspects.

In the first part of this thesis, we develop the necessary theoretical framework. Starting with a short consideration of different kinds of models in Chapter 2, we show the state of the art with respect to Lindenmayer systems in Chapter 3 and with respect to graph grammars in Chapter 4. In Chapter 5, we develop the formalism of relational growth grammars as a variant of graph grammars. We show that this formalism has a natural embedding of Lindenmayer systems which keeps all relevant properties, but represents branched structures directly as axial trees and not as linear chains with indirect encoding of branches.

In the second part, we develop the main practical result, the XL programming language as an extension of the Java programming language by very general rule-based features. Chapter 6 develops and explains the design of the language in detail; this is the main reference to get familiar with the language. Short examples illustrate the application of the new language features. Chapter 7 describes the built-in pattern matching algorithm of the implemented

run-time system for the XL programming language. Chapter 8 sketches a possible implementation of an XL compiler.

The third part is an application of relational growth grammars and the XL programming language. At the beginning of Chapter 9, we show how the general XL interfaces can be customized for relational growth grammars. On top of this customization, several examples from a variety of disciplines demonstrate the usefulness of the developed formalism and language to describe plant growth, especially functional-structural plant models, but also artificial life, architecture or interactive games. The examples of Chapter 9 operate on custom graphs like XML DOM trees or scene graphs of commercial 3D modellers, while the examples of Chapter 10 use the 3D modelling platform GroIMP, a software developed in conjunction with this thesis.

Finally, the discussion in Chapter 11 summarizes and concludes this thesis. It ends with an outlook for possible future research.

Appendix A gives an overview of the GroIMP software. The practical usage of its plug-in for relational growth grammars is illustrated in Appendix B.

Contents

Part II The XL Programming Language

1

Introduction and Motivation

Plant modelling and modelling in general try to create an image of reality in terms of some description language. Usually, the image is an abstraction and idealization of reality, but nevertheless, in most cases the long-term objective is to reach reality as closely as possible. A diversity of description languages have been used: Historically, natural languages and drawings were the first to be used, but later on, more precise specifications were expressed in terms of mathematical equations. Starting with the advent of the computer, programming languages have played a more and more important role in modelling. Likewise, there exists a diversity of models: *Qualitative* models describe observed structures and relations without statements on measured values, while *quantitative* models aim at the description of numerical values which come close to measured values [106]. *Descriptive* or *empirical* models are based on measurements and try to reconstruct the latter out of a suitable, compact specification, while *mechanistic* or *conceptual* models try to explain measurements based on general underlying mechanisms [106, 154]. *Static* models describe an image of reality at a particular point in time, while *developmental* models specify the image by a development in time [154]. Models may be simple like a parameterized function which relates variables of the model and has to be fitted to measurements (a quantitative empirical model), or complex like a system of differential equations (a quantitative mechanistic model) or an algorithmic description of growth.

Often, a model does not explicitly describe the relevant properties of a structure. It rather gives instructions how to obtain these, and the used language defines the semantics of the instructions, i. e., how the instructions are to be processed. For example, a system of differential equations in the mathematical language has to be integrated, an algorithmic description has to be executed. This execution of a model is called *simulation*, and computers are particularly qualified for this task.

Given a concrete field of application, models can not only be categorized according to the above general dimensions, but also with respect to their resolution and the considered components. In the context of plant modelling, we

can arrange models in the triangle shown in Fig. 1.1 [104]. Models of whole forest areas, landscapes or ecosystems have a very aggregated view and are typically expressed in a statistical way. When moving to scales with higher resolution, two different aspects of plants and their growth appear: the topological and geometric structure of individual plants and their components becomes visible, but also low-level biological processes with causal and functional relationships. Ideally, such a hierarchy of models has the property that the aggregation of models of higher resolution substantiates and explains the model of lower resolution. Thus with increasing biological knowledge and computational power, an obvious objective is to explain more and more scales by their underlying mechanisms, although there is not yet a consensus that this traditional method of physics is appropriate for biology [154].

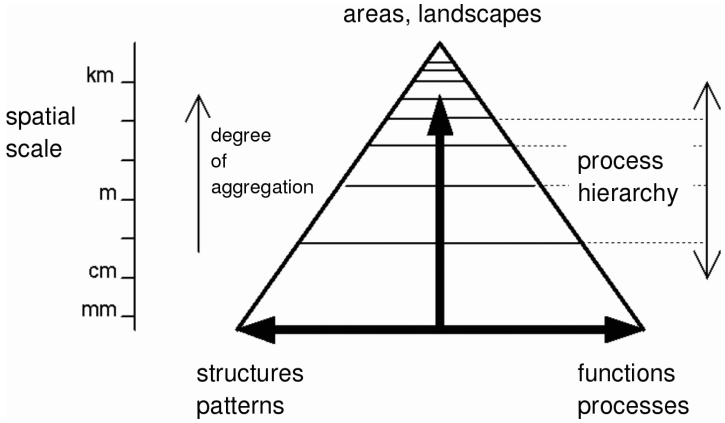


Figure 1.1. Triangle of plant models (from [104])

Historically, the two aspects at the scale of plant components, their structure and their function, were treated separately by different models (and different research groups). *Structural* models (also called *architectural* models) were mostly descriptive and did not take into account internal processes. *Functional* models (also called *process* models) focused on a detailed representation of mathematically coupled quantities of internal processes (e. g., photosynthesis, water, sugar and nutrient transport, allocation), but had only a coarse representation of structure, if any. A major reason for the simplification of structure was its irrelevance in the end: a model had to predict properties like the size of utilizable wood of a tree or the yield of a cereal. But even if only these quantities are of interest in the end, a model can benefit from the consideration of structure [104]. For example, a precise computation of photosynthesis or matter exchanges requires an accurate three-dimensional representation of the geometry. Vice versa, a descriptive structural model can benefit from the inclusion of a functional model: if one integrates processes

like photosynthesis, transport and allocation which influence the growth of structure, the model becomes a mechanistic one.

During the last decade and enabled by sufficiently powerful computers, increasing attention was drawn to such combined *functional-structural plant models* (FSPM, also called *virtual plants*) [69]. Like structural models, they represent the plant as a topological structure of interconnected components and with an associated distribution in 3D space, but now, given this detailed spatial resolution, also the specification of biological processes has to take into account the spatial distribution. As a consequence, functional-structural plant models are based on scaling up, i. e., they explain the behaviour on higher levels by underlying mechanisms. Having a detailed spatial description, interactions with the environment can be specified as this typically happens at the surface of plant organs. This includes light interception and mutual shading, but also wind attenuation and gas exchange. Having a detailed topological description, internal fluxes (water, carbon, nutrients) and signals can be defined, but also biomechanical aspects can be studied. Contrary to typical simulations of transport and mechanics in physics, the structure in which these processes happen changes itself when the plant is growing, so that there is a mutual feedback between structure and function. In general, such systems are called *dynamical systems with dynamical structure* [66].

According to [69], functional-structural plant modelling faces three challenges. At first, the modelled systems themselves have a high degree of complexity, given the numerous processes and their interactions at different scales, and the resulting variability of growth. Secondly and as a consequence thereof, the biological knowledge is very diverse and spread over several spatial and temporal scales, but this diversity has to be integrated in a single consistent model. Finally, to deal with the complexity of the systems in question, suitable formalisms and languages are needed which assist in the specification and execution of models.

This work is devoted to the latter challenge, the development of a suitable formalism and programming language for functional-structural plant modelling. The emphasis is on *suitable*: of course, any current high-level programming language can be used to implement both functional and structural aspects, but this typically amounts to a lot of code of technical nature which obscures the proper model within the implementation. An analogous example in a mathematical context is the (approximate) solution of a nonlinear equation. In a language which allows to specify such an equation in mathematical notation and which has a built-in semantics to find a root of the equation, the implementation of the model coincides with its natural specification. Contrary, the implementation in a general-purpose language requires both the implementation of the equation, typically in a non-natural way as, e. g., partial derivatives also may have to be provided, and the implementation of the solver. The essence of the original model is lost in such an implementation.

But what is a suitable programming language for functional-structural plant modelling? As in the mathematical example of nonlinear equations, the

programming language should be close to the "natural" language of plant models. Unlike in mathematics, such a standard language for all aspects does not exist, but for individual aspects, we can find more or less accepted or intuitive notations. For example, consider Fig. 1.2. It shows two growth rules at organ level. The left one describes how a terminal bud evolves into an internode together with a new terminal bud and a smaller lateral bud, i. e., it models the apical growth process which creates an axis of linearly connected internodes. The right rule describes how a lateral bud develops into a branch consisting of an internode with again two buds. The left rule also graphically describes an alternate phyllotactic pattern of branches, i. e., consecutive branches point to alternating sides of the bearing axis.

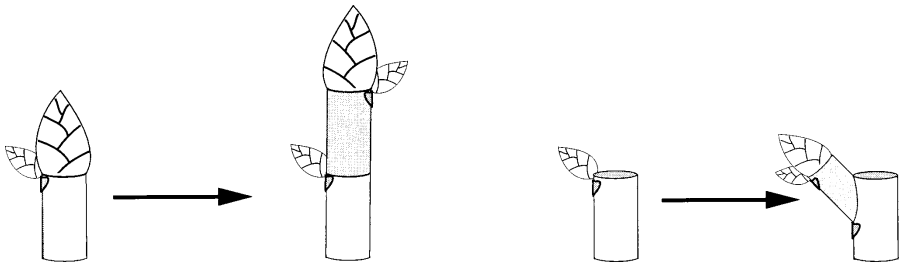


Figure 1.2. Growth rules at organ level (from [67])

These growth rules constitute a purely structural model. For their precise representation, a suitable rule-based language was already found, namely the language of *Lindenmayer systems* (L-systems for short) [161]. The structure is represented by a sequence of symbols, one for each organ (or whatever the basic structural components are). Branches are enclosed by square brackets, rotations are encoded by special rotation symbols. The right-hand side of the left rule would be written as $\mathsf{I}\ [+\mathsf{B}] \leftrightarrows \mathsf{I}\ [+\mathsf{B}] \leftrightarrows \mathsf{B}$, if I stands for an internode, B for a bud, $+$ for a rotation according to the branching angle and $\leftrightarrows$ for a rotation of 180 degrees around the growth axis according to the phyllotactic pattern. The dynamics within L-systems is implemented by replacement rules for symbols. For the two depicted rules, we have a unified single L-system rule if we observe that the lower internode in both graphical rules remains unchanged:

$$\mathsf{B} \rightarrow \mathsf{I}\ [+\mathsf{B}] \leftrightarrows \mathsf{B} \tag{1.1}$$

Figure 1.3 on the facing page shows the result of the growth rule when applied six times with an initial structure of a single bud B and with an additional scaling factor for lateral buds. The rule-based notation is very precise and concise for the description of growth of general branching structures, and it is ready to be used as a basis for a programming language. In fact, several L-system-based programming languages exist (Chap. 3), and they have served to

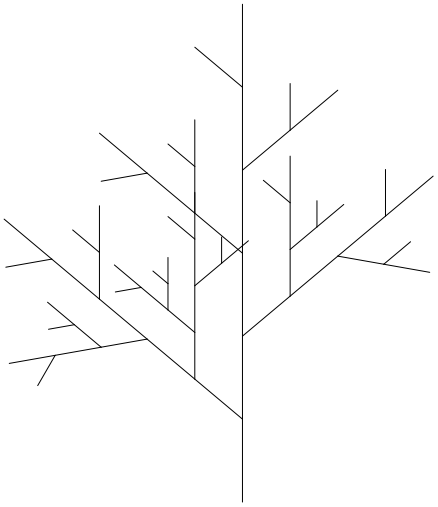


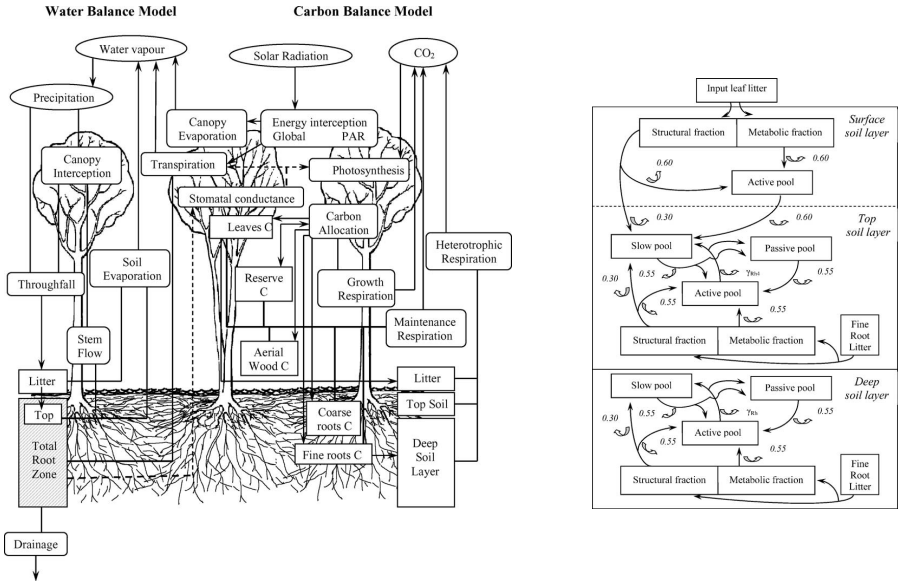
Figure 1.3. Branching structure resulting from growth rule (1.1)

implement a lot of structural plant models, but also true functional-structural plant models.

On the other hand, due to the restriction to linear sequences of symbols, the real topology of the plant can only be obtained indirectly by considering square brackets. If no interactions between components are required as in typical pure structural models, this poses no problem, but this condition does not hold for functional-structural plant models which expand functional models to the structure by means of, e. g., transport and signalling. A natural solution is to use trees or even graphs instead of linear sequences of symbols as fundamental data structure, and this is the solution which this work proposes.

Moving from sequences of symbols to trees or graphs entails a transition from L-systems to formalisms defined on such structures. However, the rule-based paradigm underlying L-systems should be retained as this is the main foundation of the success of L-systems in structural plant modelling. The corresponding formalism on the basis of graphs is given by *graph grammars*. This works develops *relational growth grammars*, a variant of graph grammars especially tailored for the use in plant modelling.

Using graphs, we have not only a suitable data structure for the topology of plants, but also for the representation of internal processes. Figure 1.4 on the next page shows typical process models for water and carbon: they use the visual language of flow diagrams, which is of course an application of the expressive power of graphs. On the other hand, this is only a qualitative description and has to be supplemented with a quantitative specification of the processes, e. g., with equations for carbon production or flow rates. This is usually done within an imperative programming language, so a combina-



(a) (b)

Figure 1.4. Flow diagrams of a process model for beech trees (from [39]): (**a**) water and carbon sub-models; (**b**) soil carbon sub-model

tion of graph grammars with imperative programming is advisable. The main practical result of this work is precisely the design and implementation of such a combined programming language, namely the *XL programming language*.

Quantitative specifications often involve differential equations, so it would be desirable to have them included in the language. This is out of the scope of this work and will be addressed in future work, however note that solvers for differential equations can nevertheless be used or implemented due to the inclusion of imperative programming.

The design of a formalism and a programming language involves an abstraction from concrete examples to common principles and requirements. The larger the considered collection of examples, the better will be the achieved abstraction. Therefore, in the following chapters we do not only take into account examples from plant modelling, but also from artificial life, i. e., from a field of study where phenomena of living systems are investigated by the consideration of (simplified) computer simulations.

Part I

The Rule-Based Paradigm

The first part of this thesis is devoted to the rule-based paradigm which, in the form of L(indenmayer)-systems, already proved to be very suitable for plant modelling. We start with a consideration of some examples of rule-based modelling from different disciplines, and study L-systems and graph grammars as two prominent and relevant representatives. Finally, we show how both can be combined to relational growth grammars, enabling the specification of functional-structural plant models and also models of other kinds such as artificial life. But what is the essence of rule-based programming? The eighth international workshop on rule-based programming (RULE 2007) characterizes this as follows [170]:

> Rule-based programming provides a framework that facilitates viewing computation as a sequence of changes transforming a complex shared structure such as a term, graph, proof, or constraint store. In rule-based languages, a set of abstractions and primitive operations typically provide sophisticated mechanisms for recognizing and manipulating structures. In a classical setting, a rule-based program consists of a collection of (conditional) rewrite rules together with a partially-explicit specification of how the rule collection should be applied to a given structure.

Thus, one has an initial structure and a set of rules, given in some rule-based language, and an execution semantics which specifies how and where to apply the (typically local) rules to the current structure, leading to a new structure. Usually, it is the task of the run-time system of the language implementation to find all locations in the structure where rules are applicable. So the programmer specifies *what* to do with the structure, possibly only under some conditions, and the computer finds *where* actions take place. Thus, the control flow is governed by the applicability of rules and the order of their processing, which typically depends on previous applications and is therefore an emergent property [107]. In systems with parallel application, there is even no order within a single transformation. This is in contrast to the imperative programming paradigm where the programmer explicitly governs the control flow and, in doing so, specifies what to change and where to do this – following the principle of a von Neumann machine.

The general advantage of rule-based modelling in the context of plant modelling and artificial life is that we typically have a large collection of individual entities (e. g., plants, plant organs, animals) which behave according to a set of common rules (e. g., growth rule for buds, movement). Using the rule-based paradigm releases the programmer from the burden to find all entities where rules are applicable, and also the application of the rules is implicit in the paradigm. This leads to a concise specification which (mostly) consists of the actual behavioural rules.

2

Introductory Examples

In this chapter, we present different kinds of rule-based modelling using simple examples. The examples serve to motivate and illustrate the presented formalisms and methods throughout the following chapters. Intentionally, they are not only related to plant modelling, but also to other fields of application in order to show the broad applicability of the rule-based paradigm. But nevertheless, also the non-plant models can be seen as metaphors of (parts of) functional-structural plant models.

2.1 Snowflake Curve

A simple but instructive example for the rule-based paradigm is the *snowflake curve* [144] which is related to the following construction of the Swedish mathematician Helge von Koch [196]:

> Joignons par une droite deux points A et B d'un plan (fig. 2.1). Partageons le segment AB en trois parties égales AC, CE, EB, et construisons sur CE comme base un triangle équilatéral CDE. Nous aurons une ligne brisée $ACDEB$ formée par 4 segments égaux. [...] Pour abréger, nous désignons par Ω cette opération au moyen de laquelle on passe d'un segment rectiligne AB à la ligne polygonale $ACDEB$ [...].
> [...]
> Effectuant l'opération Ω sur chacun de ces nouveaux segments et continuant ainsi indéfiniment, nous obtenons une suite indéfinie de lignes polygonales que nous désignerons par
>
> $$P_1, P_2, P_3, \ldots, P_n, \ldots$$
>
> et qui se composent respectivement de
>
> $$1, 4, 4^2, \ldots, 4^{n-1}, \ldots$$

côtés. P_1 désigne la droite primitive AB, P_2 la ligne $ACDEB$ et ainsi de suite.
Nous allons voir que, quand n croît indéfiniment, P_n tend vers une courbe continue P qui ne possède, en aucun point, de tangente déterminée.

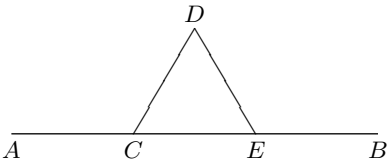


Figure 2.1. Illustration in Koch's article

Thus, von Koch's original intent was to define a continuous, but nowhere differentiable curve. In the meantime, *how* this is done has received special attention: the construction is based on the operation Ω which can be seen as a rule that, applied to a structure consisting of straight line segments, replaces every line segment AB by a polygonal line $ACDEB$. A whole family of *generalized Koch constructions* [144] exists which is based on this simple rule-based principle of *initiator* and *generator*. The initiator α (line AB in the example) represents an initial structure which is composed of a set of straight line segments. The generator Ω is recursively applied to the structure and replaces each line segment in the structure by a set of new line segments, thereby ensuring that the endpoints of a line segment and its replacement match.

What has been put in words by von Koch can also be depicted graphically:

Using a straight line as initiator α, we obtain the Koch curve as the limit $\lim_{n\to\infty} \Omega^n(\alpha)$. If α is an equilateral triangle, the snowflake curve results in the limit, see Fig. 2.2.

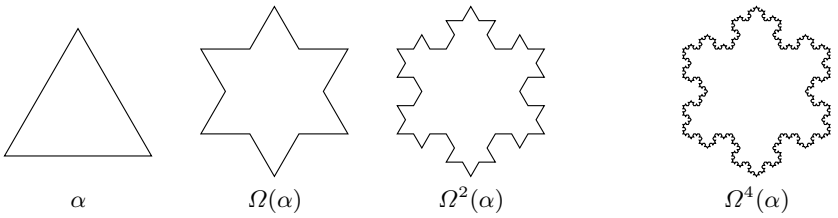
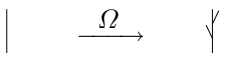


Figure 2.2. First approximations of snowflake curve

2.2 Plant-Like Branching Structure

A generalized Koch construction which replaces a straight line segment not by a linear sequence, but by a branched structure of such segments is given by the following generator from [161]:



Applied to a straight line segment as initiator, we obtain the development of a branching structure which resembles a simple herbaceous plant, see Fig. 2.3.

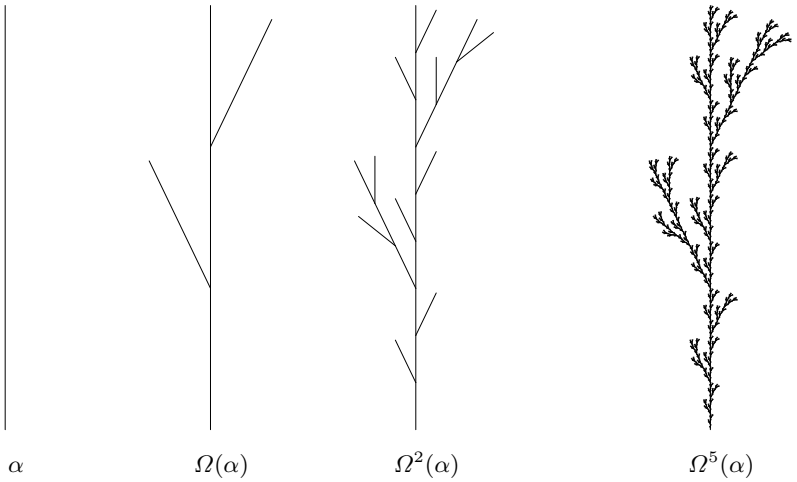


Figure 2.3. Generalized Koch construction for a branching structure

2.3 Sierpinski Triangle

The *Sierpinski triangle* can also be seen as the result of a generalized Koch construction, but with triangles instead of straight line segments. The initiator is a black triangle, the generator replaces such a black triangle by three black triangles as follows:

Figure 2.4 on the next page shows the first steps of the construction of the Sierpinski triangle.

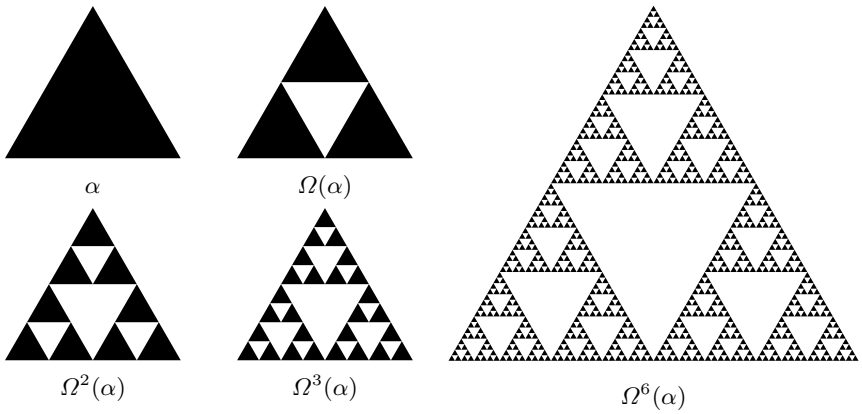


Figure 2.4. First approximations of Sierpinski triangle

2.4 Game of Life

Cellular automata [198] consist of regular, finite-dimensional grids of cells. Each cell is in one of a finite number of states. A single transition rule is applied in parallel to all cells and determines the new state of a cell as a function of the old state of the cell and its neighbours. The finite neighbourhood is defined relative to the grid position of a cell.

A famous example for a cellular automaton is the Game of Life [63]. It is defined on a two-dimensional grid with two states ('live' and 'dead') and Moore neighbourhood, i. e., the complete ring of eight neighbours (horizontal, vertical, diagonal). The transition rule is as follows:

1. A live cell with more than three or less than two live neighbours dies.
2. A dead cell with exactly three live neighbours comes to life.
3. Otherwise, the state remains unchanged.

A lot of interesting pattern evolutions can be observed. As an example, the glider pattern shown in Fig. 2.5 moves diagonally.

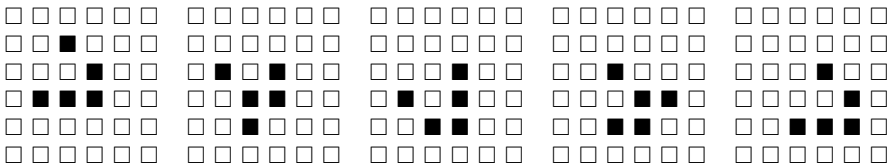


Figure 2.5. Evolution of glider pattern

The Game of Life is also a good example to illustrate the benefits of rule-based modelling. In a suitable rule-based programming language, the specification of the transition rule is a direct translation of the three lines of verbal specification from above, without any additional code. Contrary, the specification using an imperative programming language additionally requires some loops and also a mechanism to simulate the parallel application. A possible implementation in the Java programming language is

```
1  boolean[][] grid = new boolean[N][N];
2  ...
3  boolean[][] old = grid.clone();
4  for (int x = 0; x < N; x++) for (int y = 0; y < N; y++) {
5      int liveNeighbours = 0;
6      for (int dx = -1; dx <= 1; dx++) for (int dy = -1; dy <= 1; dy++) {
7          if ((x + dx >= 0) && (x + dx < N)
8              && (y + dy >= 0) && (y + dy < N) && old[x + dx][y + dy]) {
9              liveNeighbours++;
10         }
11     }
12     if (old[x][y] && ((liveNeighbours > 3) || (liveNeighbours < 2))) {
13         grid[x][y] = false;
14     } else if (!old[x][y] && (liveNeighbours == 3)) {
15         grid[x][y] = true;
16     }
17 }
```

Obviously, the essence and simple nature of the Game of Life is hidden by this implementation: the transition rule in lines 12–16 is embedded in code which iterates through the whole grid and counts live neighbours in the context. A suitable rule-based language provides both the iteration through the structure and the evaluation of context as built-in features. For an example, see Sect. 10.1.3 on page 275.

2.5 Artificial Ants

Ants are a popular subject in artificial life. 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: Short paths between, e. g., food sources and the nest are found and then followed very quickly [28].

The abstraction from the observed behaviour to a model for artificial ants leads to *agent-based modelling* [124]. Agent-based models have, among others, the following properties: A collection of individual agents act in a common environment. The behaviour is autonomous, i. e., agents make independent decisions. Agents have the capability to communicate by some protocol, and

to sense the environment. The Game of Life can be seen as a precursor of an agent-based model. The agents are live cells which make independent decisions according to the transition rule and using a simple communication to determine the number of live neighbours.

In an agent-based model of artificial ants, the environment could be a discretized 2D world just as for the Game of Life, but now each cell has a state which represents the pheromone intensity at its locations. An ant agent moves along the grid, guided by the strategy to move to the neighbouring cell with highest pheromone intensity, but to avoid previously visited cells. When moving to a cell, an agent intensifies the amount of pheromone, so the communication among agents uses the environment as medium. To mark points of interest such as food sources, ant agents get excited when on such points, which increases the amount of pheromone laid down on each movement. Although an individual ant has poor capabilities as it can only see the direct neighbourhood and has only a very restricted memory, the collective behaviour of a set of ant exhibits the remarkable emergent behaviour that short paths through points of interest are found (see Sect. 10.3.2 on page 287).

2.6 Comparison of Examples

The Koch and generalized Koch constructions create geometric shapes in 2D space and are not bound to some grid-like discretization. Contrary, the Game of Life has a fixed 2D shape consisting of the grid of cells, and it is only the states which change based on neighbourhood information. The sketched ant model is similar, but in addition to state changes, agents may also move along the grid. As we will see in the following chapters, L-systems are well suited for models of the first kind, which includes structural plant models as the main field of application of L-systems, but fail for models of the second kind with graph-like neighbourhood relations. Graph grammars provide a natural solution to this problem.

3

L-Systems

3.1 Introduction

Lindenmayer systems (L-systems for short) are the prevailing rule-based formalism in structural plant modelling. This string-rewriting formalism was introduced in 1968 by Aristid Lindenmayer, a theoretical biologist [117]. His original intent was to describe the cell division pattern of bacteria, but since this first usage the capabilities of L-systems have been explored in a variety of fields of application: most dominantly and successfully within the scope of plant modelling, but also for the construction of space-filling curves, for the generation of musical scores [70], and for architecture [70]. An excellent book on L-systems from a practical point of view has been authored by Prusinkiewicz and Lindenmayer himself [161]. The theoretical basis and some theoretical results are summarized in [90], for a collection of research papers see [168] and [169]. Foundations and application of L-systems are summarized in the sequel to an extent which is needed for the following chapters, for more details the reader is referred to the mentioned literature.

As a string-rewriting formalism, L-systems operate on a string of symbols (a word) by replacing substrings with other strings according to a set of productions. The crucial difference to other string-rewriting formalisms such as Chomsky grammars is the application of productions in parallel, i. e., every symbol is rewritten at each step of the rewriting process. The motivation for this parallelism lies in biology: cell division and other growth processes in nature happen in exactly this way. Another difference which has a more theoretical impact is that there is no distinction between terminal and nonterminal symbols.

The formal definition of the simplest class of L-systems, context-free L-systems, looks as follows. Given an alphabet (i. e., a set of symbols) V, we use the usual notations V^* for the set of all words over V (i. e., of strings of symbols) and V^+ for the set of all non-empty words over V.

Definition 3.1 (context-free L-system). *A* context-free L-system *(*0L-system *for short)* $\mathcal{G} = (V, \alpha, P)$ *consists of an alphabet* V*, an initial non-empty word* $\alpha \in V^+$ *called the* axiom *or* start word*, and a set of* productions $P \subseteq V \times V^*$*. A production* $(a, \chi) \in P$ *is written as* $a \to \chi$*,* a *is called the* predecessor *of the production,* χ *the* successor*. For all symbols* $a \in V$*, there has to be at least one production in* P *having* a *as predecessor. If there is exactly one such production for each symbol, the L-system is* deterministic *(*D0L-system *for short), otherwise it is* nondeterministic*.*

Productions are also called rules in the literature.

Definition 3.2 (direct derivation, generated language). *Let* $\mathcal{G}$ *be a 0L-system as above and* $\mu = a_1 \dots a_n$ *be a word of* n *symbols* a_i*. If there exist words* $\chi_1, \dots, \chi_n \in V^*$ *such that* $a_i \to \chi_i \in P$*, then there is a (direct)* derivation *denoted* $\mu \overset{\mathcal{G}}{\Rightarrow} \nu$ *from* μ *to* $\nu = \chi_1 \dots \chi_n$ *within* $\mathcal{G}$*. The reflexive and transitive closure of* $\overset{\mathcal{G}}{\Rightarrow}$ *is denoted by* $\overset{\mathcal{G}}{\Rightarrow}^*$*. The* language generated by $\mathcal{G}$ *is* $L(\mathcal{G}) = \left\{ \nu \in V^* \,\middle|\, \alpha \overset{\mathcal{G}}{\Rightarrow}^* \nu \right\}$*, i. e., the set of all words which can be generated from the axiom by a finite number of direct derivations.*

As for other grammars, an L-system can be used for two purposes: firstly to decide if a given word belongs to the language generated by the L-system, secondly to generate words out of the axiom. In practical applications of L-systems, the latter, generative aspect dominates. For example, within the scope of plant modelling the sequence of derived words may represent growth stages of a plant simulation.

To give a simple example for a deterministic 0L-system, let alphabet and axiom be given by $V = \{\mathsf{A}, \mathsf{B}\}$, $\alpha = \mathsf{A}$, and the set of productions P by the following two productions:

$$\mathsf{A} \to \mathsf{BA}\,,$$
$$\mathsf{B} \to \mathsf{A}\,.$$

The first derivation applies $\mathsf{A} \to \mathsf{BA}$ to the axiom A, resulting in the word BA. The second derivation simultaneously applies $\mathsf{B} \to \mathsf{A}$ to B and $\mathsf{A} \to \mathsf{BA}$ to A, resulting in the word ABA:

$$\mathsf{A} \overset{\mathcal{G}}{\Rightarrow} \mathsf{BA} \overset{\mathcal{G}}{\Rightarrow} \mathsf{ABA} \overset{\mathcal{G}}{\Rightarrow} \mathsf{BAABA} \overset{\mathcal{G}}{\Rightarrow} \mathsf{ABABAABA} \overset{\mathcal{G}}{\Rightarrow} \dots\,.$$

There are languages which can be generated by D0L-systems, but not by sequential context-free grammars. For example, consider the following D0L-system $\mathcal{G}$:

$$\alpha = \mathsf{AAA}\,,$$
$$P = \{\mathsf{A} \to \mathsf{AA}\}\,.$$

Obviously, the language which is generated by this L-system is $L(\mathcal{G}) = \{\mathsf{A}^{3 \cdot 2^i} | i \in \mathbb{N}_0\}$. This language cannot be obtained by sequential context-free

rewriting [90]. If we simply treated the L-system as a (sequential) context-free grammar $\mathcal{G}'$, we would obtain the language $L(\mathcal{G}') = \{\mathsf{A}^i | i \in \mathbb{N}, i \geq 3\} \supsetneq L(\mathcal{G})$. The effect of parallelism can be simulated by compound derivations which consist of several sequential steps in succession, but this requires an additional synchronization control mechanism which ensures that in each compound derivation for every symbol of the original word exactly one sequential derivation is applied. On the other hand, the behaviour of the sequential context-free grammar $\mathcal{G}'$ can be simulated easily by a nondeterministic 0L-system if one simply adds the identity production $\mathsf{A} \to \mathsf{A}$ to the productions P of $\mathcal{G}$.

3.2 Turtle Interpretation of Symbols

Words generated by L-systems are abstract in themselves, meaning that they have no representation in terms of a common concept scheme. However, we can choose such a concept scheme and associate its terms with the symbols in a word. Such an association is called *interpretation*. For example, the application of L-systems to create musical scores [70] has to establish a link between L-system symbols and terms of the concept scheme of music, among them notes, rests and tempo.

The prevailing interpretation of L-system symbols is the *turtle interpretation* which links symbols with two- or three-dimensional geometry. The name refers to the notion of a turtle [1]. The turtle is a metaphor for a drawing device which maintains a *turtle state* consisting of the current position and orientation in two- or three-dimensional space. The device understands a set of *turtle commands* which modify the state and/or create graphics primitives using the current state. It is common practice to use the symbol F as the line drawing command, i. e., the turtle draws a line of unit length starting at the current position in the current moving direction and then changes the current position to be at the end of the line. One can define further symbols which represent rotations, movements without line drawing, or other primitives such as circles, spheres or boxes. When the turtle is presented with the current word of a derivation step of an L-system, it reads the symbols of the word from left to right and performs the associated commands. If a symbol has no associated command, it is simply ignored.

As an example, let us consider the implementation of the snowflake curve (Sect. 2.1 on page 11) by an L-system. Naturally, the axiom of the L-system has to represent the initiator, namely the initial equilateral triangle, and the set of productions P has to contain a translation of von Koch's generator Ω. If we use the symbol F to denote a line, each side of the initial triangle and each straight line within the generator production can simply be represented by an F. We further need turtle commands for rotation in order to implement the angle of 120 degrees of the equilateral triangle and the angles of 60 and 120 degrees of the generator. Thus, let the symbols Left and Right command the

turtle to rotate to the left or right, respectively, by 60 degrees. Note that these symbols are treated as individual symbols and not as sequences of single-letter symbols. The L-system translation can then be formulated as follows:

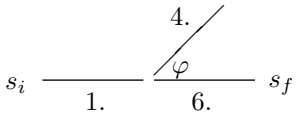
$$\alpha = \mathsf{F\ Right\ Right\ F\ Right\ Right\ F}\ ,$$
$$\mathsf{F} \rightarrow \mathsf{F\ Left\ F\ Right\ Right\ F\ Left\ F}\ .$$

In this and all following examples, for every symbol a for which there is no explicit production having a as predecessor the identity production $a \rightarrow a$ has to be added (in this case $\mathsf{Left} \rightarrow \mathsf{Left}$ and $\mathsf{Right} \rightarrow \mathsf{Right}$). The turtle interpretation of the results of derivation yields approximations of the snowflake curve as in Fig. 2.2 on page 12.

Obviously, the sequence of symbols in a word is linear. Thus, without further arrangements the turtle interpretation yields a geometric structure which consists of a chain of geometric primitives stacked on top of each other. While this is suitable for the snowflake curve, branched topologies cannot be represented this way. But of course, the capability of representing such branched topologies is an important prerequisite for the modelling of plants and other non-linear structures.

The restriction to non-branched topologies can be overcome by the inclusion of two special turtle commands: the first one pushes the current turtle state onto a stack which is exclusively used for this purpose, the second one pops the top from this stack and restores the current turtle state to the popped value. The bracket symbols [and] are customarily used to denote the push- and pop-operations, and L-systems which make use of them are called *bracketed L-systems*. For example, the word F [$\mathsf{Left\ F}$] F commands the turtle to perform this sequence of operations:

1. Draw a first line.
2. Push current state onto top of stack.
3. Rotate to the left by a predefined angle φ.
4. Draw a second line. This line starts at the end of the first line at an angle of φ.
5. Pop state from stack so that it equals the state before 2.
6. Draw a third line. This line extends the first line straight.



The figure shows the resulting branched structure, s_i and s_f indicate the initial and final turtle state, respectively.

A further special turtle command is the *cut-operator*, this command is usually denoted by %. It sets a flag in the turtle state which indicates that the following turtle commands shall be ignored with the exception of the push- and pop-commands. The effect is that the current branch is cut off up to its terminating]-symbol.

Now let us have a look at a more interesting example of a growing binary tree. Its leaves (in the sense of graph theory) are represented by the symbol Bud which is the axiom and predecessor of the single growth production:

$$\begin{aligned} &\alpha = \mathsf{Bud}\ , \\ &\mathsf{Bud} \rightarrow \mathsf{F}\ [\mathsf{Left\ Twist\ Bud}]\ [\mathsf{Right\ Twist\ Bud}]\ . \end{aligned} \tag{3.1}$$

In order to extend into three dimensions, the symbol Twist commands the turtle to rotate about its own axis by 90 degrees. A visualization of the outcome is shown in Fig. 3.1, where Bud symbols are drawn by small squares and the angle of rotation of Left and Right is 30 degrees.

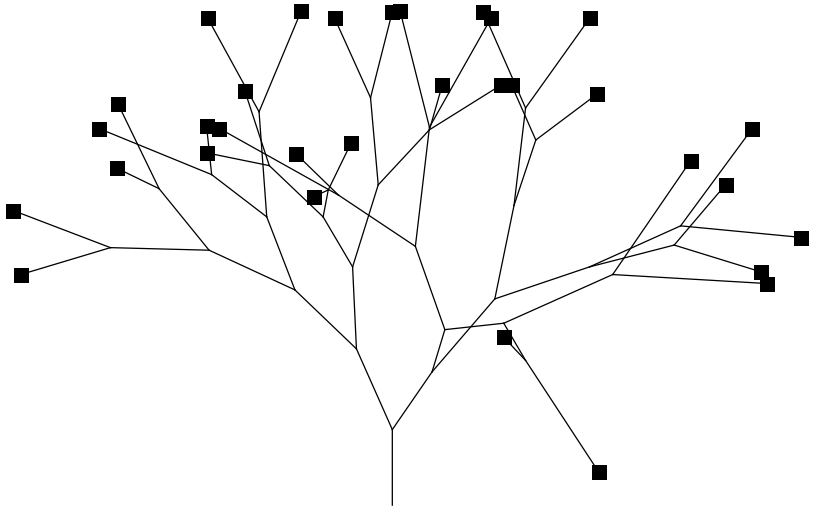


Figure 3.1. Turtle interpretation of L-system (3.1) after six derivation steps

With the help of the bracket symbols, we can also model the Sierpinski construction (see Sect. 2.3 on page 13) by an L-system. If T stands for a "germ" of a triangle, we have to replace such a germ by an actual triangle consisting of three F commands with rotations (thus triangles are not filled, but represented by their outline), and by three new germs for the next generation. We also have to ensure that the length of triangle edges of a generation is half of this length of its predecessor generation. This is simply done by doubling the length of existing triangles, namely by replacing each existing F by two F commands.

$$\begin{aligned} &\alpha = \mathsf{T}\ , \\ &\mathsf{T} \rightarrow \mathsf{T}\ [\mathsf{F\ Left\ T\ F\ Left\ T\ F}]\ , \\ &\mathsf{F} \rightarrow \mathsf{F\ F}\ . \end{aligned} \tag{3.2}$$

The angle of rotation of Left is 120 degrees. After six steps, we obtain the visualization depicted in Fig. 3.2. Note that although the visualization matches the result of the Sierpinski construction, the internal structure created by the L-system, namely a string with brackets, can be topologically interpreted only as a tree. This means that connectivity information between segments is not completely represented by the L-system string. Some vertices of triangles simply happen to be at the same location in 2D space without being topologically linked in the string.

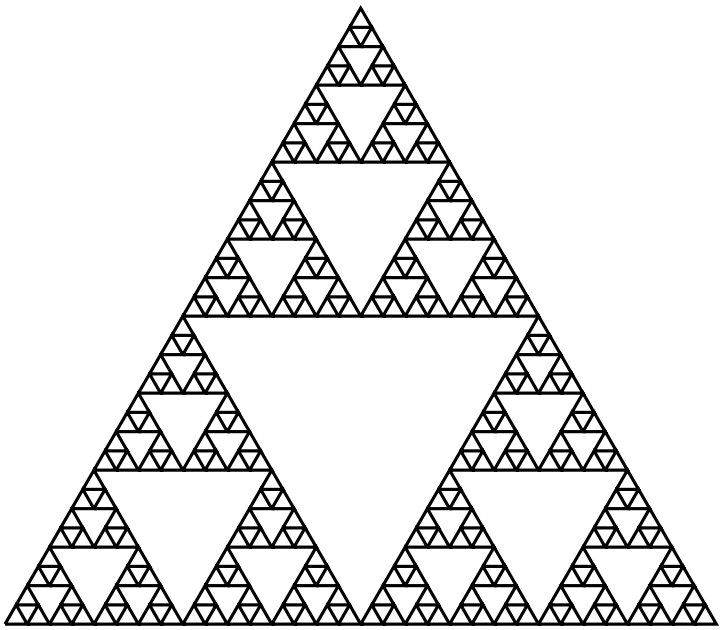


Figure 3.2. Turtle interpretation of L-system (3.2) after six derivation steps

3.3 Stochastic L-Systems

If one uses a nondeterministic L-system in the generative way, i. e., if a sequence of words shall be obtained by repeated derivation starting with the axiom, there is a natural, stochastic solution how to handle the nondeterminism: on derivation of a predecessor for which there are alternative productions, one of the alternatives is chosen (pseudo-)randomly according to given probabilities. The probabilities have to be specified along with the productions, leading to the notion of a stochastic L-system:

Definition 3.3 (stochastic 0L-system). *A* stochastic 0L-system $(\mathcal{G}, \pi)$ *is an 0L-system* $\mathcal{G}$ *together with a* probability function $\pi : P \to (0, 1]$ *such that for each symbol* $a \in V$ *the sum* $\sum_{\{\chi \in V^* | a \to \chi \in P\}} \pi(a \to \chi)$ *equals one.*

Definition 3.4 (stochastic derivation). *Let* $(\mathcal{G}, \pi)$ *be a stochastic 0L-system. A* stochastic derivation $\mu \overset{(\mathcal{G},\pi)}{\Longrightarrow} \nu$ *is a derivation such that for each occurrence of a symbol* a *in* μ *the probability of applying the production* $a \to \chi \in P$ *is given by* $\pi(a \to \chi)$.

The nondeterminism of stochastic L-systems can be used to model processes for which an underlying deterministic mechanism cannot be specified. For example, consider a plant model where lateral buds in leaf axils may either grow out, remain dormant or die. If there is a lack of knowledge of the underlying causes, or for the sake of simplicity, one can just assign probabilities to the three types of bud development which reflect their observed frequencies. Such a technique is used by the following L-system which extends L-system (3.1) on page 21. Figure 3.3 displays a visualization.

$$\begin{aligned} &\mathsf{Bud} \xrightarrow{0.2} \mathsf{F}\ [\mathsf{Left\ Twist\ Bud}]\ [\mathsf{Right\ Twist\ Bud}]\ , \\ &\mathsf{Bud} \xrightarrow{0.7} \mathsf{Bud}\ , \\ &\mathsf{Bud} \xrightarrow{0.1}\ . \end{aligned} \tag{3.3}$$

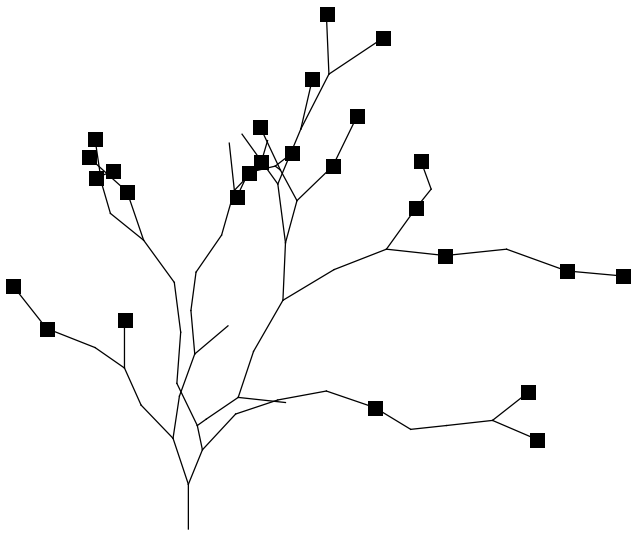


Figure 3.3. Turtle interpretation of L-system (3.3) after some stochastic derivation steps

3.4 Context-Sensitive L-Systems

The L-systems considered so far are context-free. In practical terms of modelling, this means that the development of each entity of the model is completely isolated from the other entities. There is no possibility of flow of information. This precludes the implementation of causal dependences which are the basis of advanced models of, e. g., plant development.

Both deterministic and stochastic 0L-systems can be augmented to *context-sensitive L-systems*. A production of a context-sensitive L-system can only be applied if the local context (one or more neighbouring symbols to the left and/or to the right) of the predecessor fulfils some conditions which are specified along with the production. In the simplest case, these conditions state that the context has to consist of specific symbols; such an L-system is called 2L-system if the context consists of one symbol to the left and one to the right. The actual treatment of context-sensitive L-systems in L-system software usually loosens the notion of neighbourhood: one may specify a set of symbols which shall be ignored in context matching, and entire subsequences enclosed in brackets are skipped so that symbols which are neighbouring in the turtle interpretation are also neighbours for context matching.

3.5 Table L-Systems

The previously considered L-systems contained a single set of productions. At each direct derivation step, the current word is subjected to all productions. However, there are situations where we have several groups of productions and only some of them should be active for a specific derivation. For example, we could divide the growth of a perennial plant into a repeated sequence of growth in spring, summer, autumn and winter. For each season, a specific group of productions is appropriate. This idea lies behind *table L-systems* which specify a list of tables of productions [90].

Definition 3.5 (T0L-system). *A* T0L-system $\mathcal{G} = (V, \alpha, T)$ *consists of an alphabet* V*, an axiom* $\alpha \in V^+$ *and a finite set* $T = \{P_1, \ldots, P_n\}$ *of sets of productions* $P_i \subseteq V \times V^*$ *such that for each* $P \in T$ *the triple* (V, α, P) *is a 0L-system.*

Definition 3.6 (direct derivation). *Let* $\mathcal{G} = (V, \alpha, T)$ *be a T0L-system and* $\mu, \nu \in V^*$*. There is a (direct)* derivation *from* μ *to* ν *if there is a* $P \in T$ *such that* $\mu \overset{(V,\alpha,P)}{\Longrightarrow} \nu$*, i. e., one of the tables of productions derives* ν *from* μ*.*

This definition does not control the order and multitude of the used tables. One can define additional means to specify the order of tables, for example graphs whose nodes define the currently active tables and whose edges define the allowed paths, or finite automata [142].

3.6 Pseudo L-Systems

Predecessors of L-system productions consist of a single symbol, it is only the context which may consist of entire words. Pseudo L-systems generalize this so that (non-empty) words are allowed even for predecessors [153]. For a direct derivation, the current word has to be partitioned completely and disjointly by predecessor words of productions. This may lead to an additional nondeterminism: for L-systems, nondeterminism is possible only at the level of individual symbols which may be replaced by different successors, while for pseudo L-systems, there may exist different partitions of the whole word. The definition of [153] circumvents this nondeterminism by two additional specifications: a direct derivation is constructed by scanning the current word from left to right, and for each position, the first applicable production of the ordered list of productions is chosen.

3.7 Parametric L-Systems

The symbols of L-systems are of a discrete nature both in time and space: the course of time is represented by discrete derivation steps, and the structure consists of symbols which are either present, meaning for example an internode of a plant, or not present. Discretization of time is quite common in modelling and simulation, because most processes cannot be integrated analytically with respect to time and, thus, have to be discretized anyway. However, the strict discreteness in space caused by plain symbols makes the corresponding L-systems unsuitable for the modelling of processes with continuous properties, for example smooth elongation of an internode. There is a simple yet powerful solution of this shortcoming, namely the addition of (real-valued) parameters to symbols and productions as in this production:

$$\mathsf{A}(x, y) \rightarrow \mathsf{A}(1.4\,x, x + y)\ . \tag{3.4}$$

Applied to the parameterized symbol $\mathsf{A}(3, 1)$, the production yields $\mathsf{A}(4.2, 4)$. A parameterized symbol is called a *module*, an L-system which makes use of such modules is a *parametric L-system* [161, 157, 74]. Every class of L-systems can be extended easily to its parametric counterpart. For formal definitions, the reader is referred to [74] and the more general approach in [23].

As indicated by production (3.4), a parameterized production has as its left-hand side a symbol and a number of formal parameters. On derivation, these formal parameters are bound to the values of the actual parameters of the module to which the production is applied. These values are then used on the right-hand side to determine the actual parameters for the replacing modules. Arithmetic expressions can be used for this purpose, consisting of standard operators and mathematical functions. Also the probability associated with a production may be an arithmetic expression.

Once that we have included parameters in L-system productions, we can also make use of them in *application conditions* which in this case are predicates (logical expressions) on the parameters. A production can only be applied if its application condition is fulfilled. This can be compared to context-sensitivity: the latter poses restrictions on the structure (namely on the neighbourhood of a symbol), while predicates pose restrictions on attributes (namely on the parameters of a module).

The following parametric L-system makes use of parameters, application conditions and context. The notation follows [161], i. e., application conditions are specified after a colon, and an optional context is separated from the predecessor by angle brackets as in $left < pred > right$.

$$\alpha = \mathsf{T}(10)\ \mathsf{Bud}(10) \tag{3.5}$$

$$\mathsf{Bud}(a) : a > 0 \rightarrow \mathsf{F}(0.2)\ [\mathsf{Left}\ \mathsf{F}(1)]\ [\mathsf{Right}\ \mathsf{F}(1)]\ \mathsf{Bud}(a-1) \tag{3.6}$$

$$\mathsf{Bud}(a) : a = 0 \rightarrow \tag{3.7}$$

$$\mathsf{T}(a) \rightarrow \mathsf{T}(a-1) \tag{3.8}$$

$$\mathsf{T}(a) < \mathsf{F}(x) : a = 0 \rightarrow \mathsf{F}(1) \tag{3.9}$$

$$\mathsf{F}(x) < \mathsf{F}(y) \rightarrow \mathsf{F}(\frac{x+2y}{3}) \tag{3.10}$$

The first two productions (3.6) and (3.7) induce a growth of structure with single-line branches. This structure is limited to ten repetitions due to the decrement of the parameter a of the Bud. Meanwhile, the counter of a timer T is decremented, too, by production (3.8). If the counter is zero, the parameter of the first F (which has T as its left neighbour) is set to one by production (3.9). The last production (3.10) implements a gradual propagation of the parameter value of a module F to its right neighbour. If a module $\mathsf{F}(x)$ is interpreted as a turtle command which draws a line of length x, we get the development of Fig. 3.4 on the facing page.

As it can be seen from this example, context-sensitive, parametric L-systems are able to propagate some continuous signal through the L-system word by means of parameter values. This flow of information is a very important feature in plant modelling, as it can be used to model signalling and transport. However note that due to the restriction to local context, the number of steps which it takes to propagate a signal along a path of N symbols is proportional to N. The speed of flow of information is thus tightly coupled with the time resolution of derivation steps.

3.8 Differential L-Systems

Parametric L-systems introduce continuity in space to L-systems by the addition of continuous parameters to the otherwise discrete structure of symbols. The sequence of derivation steps leads to an inherent discretization of time

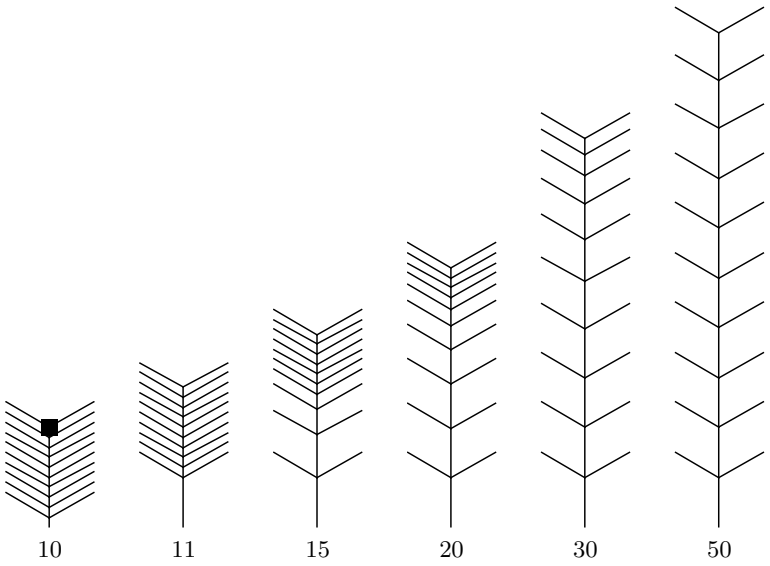


Figure 3.4. Development of L-system (3.5)–(3.10) after the indicated number of derivation steps. At step 11, elongation is triggered. In the following steps, this propagates through the structure

which cannot be overcome as easily. As has been said in the previous section, this usually poses no problems because a discretization of time is needed anyway. But if this discretization is only due to a numerical solver for a differential equation, one can equip the formalism of L-systems itself with differential equations and thus dispense with the need for an implementation of the solver within the L-system.

The notion of a *differential L-system* (dL-system for short) is defined in [156]. Ordinary differential equations with respect to time are associated with the parameters of modules, these equations may include parameters of context. The resulting development in time is C^1-continuous until the trajectory of a parameter vector of a module reaches some boundary which has been specified along with the differential equation: this event triggers the application of a production to the module and may therefore result in a discontinuity in both topology and parameter values. Thus, topological changes, which are discrete by their very nature, happen at discrete points in time by means of application of productions; in between a C^1-continuous development of parameter values is governed by differential equations.

Of course, a concrete implementation of a dL-system on a computer has to solve the differential equations by means of a numerical algorithm which necessitates a discretization in time (unless the differential equation can be solved analytically, which is a very rare case). But this discretization due to

technical demands can be chosen independently of any model-inherent discretization. A good solver will choose it as fine as needed to guarantee the requested accuracy.

3.9 Interpretive Productions

Throughout the previous examples, the symbol F (or the module $\mathsf{F}(x)$) has been used as turtle command to draw a line. For the tree-like examples, we can also think of them as simple botanical models and attribute the botanical meaning of an internode to F. Thus, for visualization purposes the symbol F stands for a geometric entity, while it stands for a botanical entity in terms of the model. This mixture of meanings often turns out to be disadvantageous, especially if the model becomes complex. In fact, it is in conflict with the principle of *separation of concerns* which is well-known from software engineering and should be followed as far as possible.

The mechanism of *interpretive productions* provides a solution to this problem within the formalism of L-systems itself [103]. Interpretive productions are represented by a second set of productions P_I. The interpretation of a current word μ is mediated by P_I: at first, a derivation of μ using P_I is performed, then the usual interpretation I (e. g., a turtle interpretation) is applied to the derived word μ'. In order to distinguish interpretive productions from the original productions of the L-system, the latter are called *generative productions*. Figure 3.5 illustrates the mechanism of L-systems with interpretive productions.

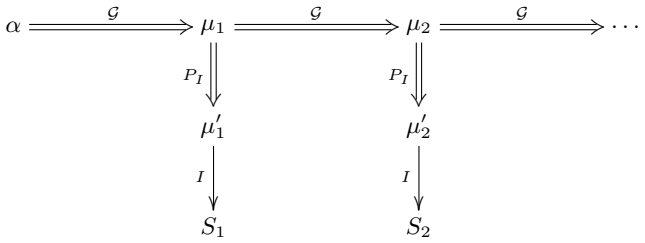


Figure 3.5. Generative ($\mathcal{G}$) and interpretive (P_I) derivations, turtle interpretation (I), resulting structure (S)

Using interpretive productions, visualization can be separated from the L-system model. For example, the model may use a module $\mathsf{I}(m)$ which stands for an internode of mass m. The interpretive production

$$\mathsf{I}(m) \rightarrow \mathsf{F}(l(m))$$

leads to a visualization of the internode as a line of length $l(m)$, where the function l reflects the allometric relation between mass and length of internodes. As another example consider a flower which is represented by a single symbol in the model, but visualized nicely by a number of turtle commands.

3.10 L-Systems with Imperative Programming Statements

Having introduced parameters, it is not unusual that the expressions in production successors and application conditions become complex and share common subexpressions. For the sake of readability (and runtime efficiency if there is no optimizing compiler), variables local to productions would be helpful in such situations [74]. They have to be assigned before they are used just as in an imperative programming language. Afterwards, their value can be used in expressions.

As the assignment of variables with later use of the values is characteristic of imperative programming, a consequential extension is to allow not only assignments, but also other statements to be executed as a side effect of production matching and application. Such an extension was developed in [74], where a block of statements is executed before the application condition is evaluated and another block before the successor is generated (if the application condition has been fulfilled). The statements consist of assignments to both local and global variables, text output to a console and control structures **if**, **while**, **do**. Additional blocks of statements are executed before the L-system derivation begins, before each derivation step, at the end of each derivation step and at the end of the entire derivation process. The following code shows such an L-system in the notation of the cpfg software (Sect. 3.15.2 on page 36):

```
1  #define STEPS 10
2
3  Lsystem: 1
4
5  /* flowering indicates if flowers should be created for internodes in
6     the current step. n is the total number of internodes, len the
7     length of internodes which are created in the current step. */
8  Start: {flowering=0; n=0; len=1;}
9  StartEach: {len=len*0.8;}
10 EndEach: {
11     printf("Number of internodes = %3f, Flowering = %3f\n",n,flowering);
12 }
13 End: {printf("Done\n");}
14
15 Seed:1
16
17 derivation length: STEPS
```

```
18
19 /* Start word. !(x) sets line width to x. */
20 Axiom: T(4) !(1) X
21
22 /* T(x) represents timer, sets flowering to 1 every fifth step. */
23 T(x) : 1 {if(x <= 0) {flowering=1; y=4;} else {flowering=0; y=x-1;}}
24     --> T(y)
25 /* X generates tree. +(a), -(a) and /(a) are rotations. */
26 X : 1 {n=n+1;} --> F(len) [+(30) /(90) X] [-(30) /(90) X] : 0.86
27 /* Probability of termination of growth at X is 14% (* = empty word)*/
28 X --> * : 0.14
29 /* If flowering has been set, create a flower for every F. */
30 F(x) : (flowering > 0) --> F(x) [Y !(10) F(0.05)]
31 /* Cut flowers created in the previous step (% is cut operator). */
32 Y --> %
33
34 endlsystem
```

It should be noted that the introduction of global variables and corresponding assignment statements conflicts with the parallelism of L-systems: different production applications may assign different values to a global variable in parallel, so the result is undefined. If a single parallel derivation is modelled by a sequence of synchronized sequential derivations as explained on page 19, the result depends on the chosen order of sequential derivation. If this is fixed, e. g., from left to right, the semantics is a well-defined mixture of parallel and sequential computation. The example makes use of this: Since the timer module is the leftmost module in the word, its code is executed at first. The computed value of `flowering` (line 23) is thus available in the applications of the flowering production in line 30 within the same derivation step.

3.11 Growth Grammars

Growth grammars [103, 106] are an extension of parametric, stochastic 0L-systems with interpretive productions. They are tailored for the needs of plant modelling, especially of tree modelling, and the software GROGRA provides an implementation which is presented in Sect. 3.15.1 on page 35. Growth grammars are equipped with a fixed turtle interpretation of modules which provides a rich set of turtle commands:

- Basic commands such as cylinder drawing (as a generalization of lines), movement and rotation about the three axes. The generated cylinders are called *elementary units*.
- Geotropism commands, i. e., rotations towards the global vertical direction.
- Commands to modify the turtle state, e. g., to set the current length or diameter, or botanical parameters of the turtle state such as the biomass.
- Object instancing command to draw a previously defined substructure.

- Commands to modify values of registers (see below).

Expressions within growth grammars can use a lot of predefined functions, among them mathematical functions and stochastic functions. A speciality of *sensitive* growth grammars are *sensitive functions*: they depend on the previously created geometric structure and compute, among others, the global coordinates of the current position, the distance to the closest neighbouring elementary unit, or the total biomass of all elementary units within a vertical cone starting at the current position (the *light cone*). These functions are *globally sensitive* since their values depend on the structure as a whole and not just on the local context of the current symbol as it is the case for context-sensitive L-systems. Fig. 3.6 illustrates this dependence. Note that now the turtle interpretation is mandatory even if no visualization is desired.

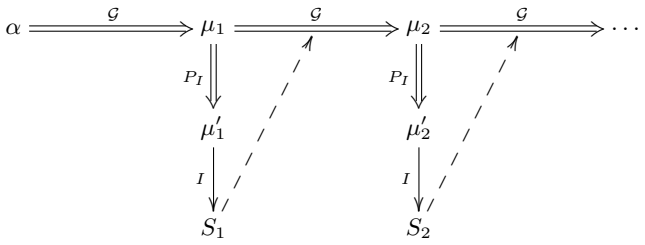


Figure 3.6. Sensitivity: dependence of derivation on structure (*dashed arrows*)

Sensitive functions can be used for a lot of purposes. A simple, botanical usage would be to make the growth dependent on the current position. This could reflect the quality of the soil, a simple radiation model based on horizontally homogeneous light extinction, some fixed obstacles to be avoided, or a prescribed overall shape as in a topiary garden. A more advanced application is the dependence of growth on the biomass in the light cone, this models the competition for light even in heterogeneous stands in a realistic way.

Arithmetical-structural operators [105] are a further feature of growth grammars. They evaluate a subexpression for every elementary unit of an operator-specific set and compute the sum of the obtained values. There are operators which compute the sum over all children of the current elementary unit, over all descendants, and over all ancestors. Thus, the computation of, for example, the total biomass of a branch including nested branches is easily specified.

Growth grammars define local and global, real-valued *registers* [110, 105]. They are modified by special command modules, and they can be used in expressions as part of parameter computations, conditions or production probabilities. Depending on the command module, the modification happens as a side-effect of turtle interpretation, or it is already performed as part of the generative derivation when the command module is created. In the latter case,

a global flow of information from a module to all other modules on the right within a single step is possible, provided that the derivation replaces modules from left to right, i. e., that it is a compound derivation consisting of sequential derivations from left to right with synchronization control as it has been explained on page 19. This is similar to the introduction of global variables which has been discussed in the previous section.

On the right-hand side of a production, a *repetition operator* is allowed which contains a sequence of modules. When such a production is applied, this operator is not inserted verbatim as part of the successor of the derivation. Instead, the operator produces a number of repetitions of its contained modules. The number of repetitions is an expression, the parameters of the produced modules may depend on the current repetition index. Thus, the structure of the successor of a production is no longer fixed, it is rather dynamically created depending on the current context.

Such a dynamic creation of the successor is also effected by the *expansion operator* [105]. Like the repetition operator, it contains a sequence of modules. But the expansion operator treats this sequence as an axiom and performs a number of direct derivations according to the L-system, where the number is specified as an expression. The final result of this derivation sequence is inserted as part of the successor.

3.12 Environmentally-Sensitive L-Systems

Similar to growth grammars, *environmentally-sensitive L-systems* [155] provide a means to include a dependence on the current global position and orientation in the derivation step of the underlying L-system. While growth grammars provide sensitive functions for this purpose, environmentally-sensitive L-systems make use of special *query modules*. When such a special module is encountered on turtle interpretation, which is now mandatory as for sensitive growth grammars, its parameter values are set to the current position or to the current turtle orientation depending on the letter of the module. These new parameter values can then be used in the next derivation step, so the flow of information is similar to Fig. 3.6 on the previous page.

3.13 Open L-Systems

Open L-systems [129] are an extension of environmentally-sensitive L-systems. The denotation *open* stresses the contrast to conventional L-systems which are closed systems without any interaction with an external environment. Open L-systems provide a means of bidirectional communication between the L-system and an environment based on an extension of the mechanism of query modules. The environment is represented as a mapping which receives data of a number of *communication modules* and maps this data to new parameter values for

these modules. The data to be received by the environment is determined during turtle interpretation: when a communication module is encountered, its identity (i. e., its position in the current word), its parameter values and the symbol and parameter values of the following module are collected along with the current turtle state. Note that the environment therefore has a restricted knowledge of the structure: any module which is neither a communication module itself nor preceded by such a module is hidden from the environment.

The environment may represent, for example, a terrain which returns its height at a given position, it may represent a model of light competition based on light cones as it is defined for growth grammars, or it may represent a physically-based radiation model (see Sect. B.14 on page 412). In any case, such a concrete environment has to be specified externally to the L-system: Open L-systems provide an abstract communication mechanism between an L-system and an environment, while sensitive growth grammars provide a set of fixed, intrinsic environmental algorithms.

3.14 L+C

In contrast to the previously described formal L-system extensions, the *L+C programming language* [91, 92, 159] is a concrete programming language which allows the specification of L-system productions by a special syntax. It has been defined on the basis of the considerations in [160] in order to meet the growing needs of functional-structural plant modelling. The L+C programming language is specified in terms of a *preprocessing step* that translates the L+C source file into a C++ source file which is then compiled by an ordinary C++ compiler. The syntax for productions and some further syntactical constructs are substituted by C++ equivalents, which are composed of a set of statements including invocations of functions of a special support library for L+C. The rest of the L+C source file is taken over verbatim, i. e., as ordinary C++ source code. Such C++ source code may even be used within the right-hand side of productions, its side effects may influence the production successor.

The advantage of such an approach is obvious: The general logic of L-systems as a rule-based formalism is retained in order to model the structure, while pieces of imperative code, specified in the well-known all-purpose language C++, can be used to modulate production application. These pieces are responsible for those parts of the model which cannot be implemented reasonably by pure L-system productions. This substantial qualitative enhancement can be put into practice by the comparatively uncomplex implementation of a source code processor, the difficult task of the compilation of the source file to efficient executable code is shifted to an existing C++ compiler.

The following code shows an example of the L+C syntax:

```
1 #include <lpfgall.h>
2 #include <math.h>
3
```

```
4  #define STEPS 10
5
6  derivation length: STEPS;
7
8  module X(float);
9  module Y(int);
10
11 float exp;
12
13 Start: {exp = 1.2;}
14
15 Axiom: SetWidth(0.1) X(1);
16
17 production:
18
19 X(t): {
20     float x = pow(t, exp);
21     nproduce F(1) SB();
22     if ((t & 1) == 0)
23         nproduce Left(60);
24     else
25         nproduce Right(60);
26     produce Flower((int) x) EB() X(t+1);
27 }
28
29 interpretation:
30
31 Flower(n): {
32     nproduce F(1);
33     for (int i = 0; i < n; i++)
34         nproduce SB() RollL(360*i/n) Left(60) F(1) EB();
35     produce;
36 }
```

Lines 8 and 9 declare the model-specific modules including their parameter types. The latter are not restricted to numerical types, any type can be chosen. The section labelled **Start** in line 13 specifies the statements which shall be executed at the beginning of the whole derivation process, this is the same mechanism as the one described in Sect. 3.10 on page 29. Symbol names of the L+C programming language are valid identifiers of the C++ programming language, the symbols `SB` (start branch) and `EB` (end branch) replace the square brackets of the usual L-system notation. The example consists of a generative production in lines 19 to 27 and an interpretive production in lines 31 to 36. Their predecessors `X(t)` and `Flower(n)` are replaced by successors which are constructed dynamically by **nproduce** and **produce** statements. While **produce** terminates the construction of the successor, **nproduce** only generates a partial result and has to be followed by further **nproduce** statements or a terminating **produce** statement. As the example shows, this dynamic con-

struction of the successor can be governed by control flow statements like **if** and **for**.

The L+C programming language also defines a *new context* which is the neighbourhood of the successor of a production in the new word. If a derivation is implemented by sequential steps from left to right, then the new context to the left is already known on application of a production, and analogously with left and right exchanged. For practical reasons, the new context is thus restricted to either side, depending on the choice of *derivation direction*. This is a special case of a *predictive context* [33] which allows both sides to be used as context. An advantage of these contexts is that information can be transferred globally in a single direct derivation. In the case of new context, the direction of information flow is restricted to the derivation direction.

3.15 L-System Software

This section presents currently available software for the execution of L-system based models. The list is not exhaustive.

3.15.1 GROGRA

The software GROGRA [103] is an interpreter for growth grammars (Sect. 3.11 on page 30). Growth grammars have to be specified in text files. The (non-sensitive) growth grammar for the snowflake curve looks as follows:

```
1 \angle 60,
2 * # F - - F - - F,
3 F # F + F - - F + F
```

`+` and `-` are rotation commands, their angle is set to 60 degrees in the first line. The second line specifies a production which replaces the default axiom `*` by the encoding of a triangle. The third line implements the generator of the Koch curve. As we can see, production arrows are represented by `#`.

The following code is an example of a sensitive growth grammar.

```
1 \var z zcoordinate,
2 * # F(100) a(100),
3 (z<250) a(s) # [ RU-60 F(s) a(s*0.7) ] RU10 F(s) d a(s*0.9) ?0.8,
4 (z<250) a(s) # RU10 F(s) d a(s*0.9) ?0.2,
5 d # RH180 ?0.5,
6 d # ?0.5,
```

The variable `z` is declared to represent the current absolute z-coordinate in line 1. Lines 3 and 4 specify stochastic productions: with a probability of 80 %, `a(s)` produces a side branch, otherwise growth continues without branching. Both productions are subjected to the condition that the z-coordinate (of the tip of the `F` preceding `a(s)`) has to be less than 250. `RU` and `RH` are rotations.

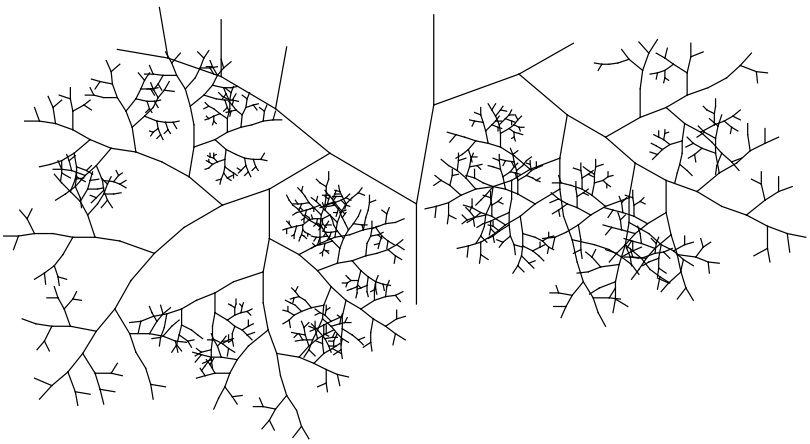


Figure 3.7. Turtle interpretation of sensitive growth grammar after 13 derivation steps

A single literal value for a parameter can be specified without parentheses immediately after the symbol as in `RU-60`. The outcome of this example is shown in Fig. 3.7 where one can recognize the horizontal growth limit `z < 250`.

The strength of GROGRA lies both in the underlying formalism of growth grammars and in the numerous integrated analysis algorithms. Growth grammars contain the unique features of globally sensitive functions and arithmetical-structural operators which make them particularly apt for the modelling of trees with geometrically caused interaction. By means of the analysis algorithms, the generated structure can be investigated in detail, for example with respect to its topology, fractal properties, or botanical properties. It is also possible to import an existing structure (e. g., a measured tree) into GROGRA, the analysis algorithms can then be used equally well for this structure.

Figure 3.8 on the facing page shows the outcome of a spruce model simulated with GROGRA [106]. It has been developed on the basis of measurements in a pure spruce stand in the Solling (German midland mountains).

3.15.2 vlab and L-Studio

vlab (virtual laboratory) and L-Studio are both based on the L-system engines cpfg and lpfg [152]. While vlab runs on UNIX systems, L-Studio is a program for Windows. The engine cpfg (plant and fractal generator with continuous parameters) implements open L-systems with imperative programming statements as described in Sect. 3.10 on page 29 and Sect. 3.13 on page 32. The engine lpfg implements the L+C programming language (Sect. 3.14 on page 33), it invokes the source code preprocessor and afterwards a C++ compiler.

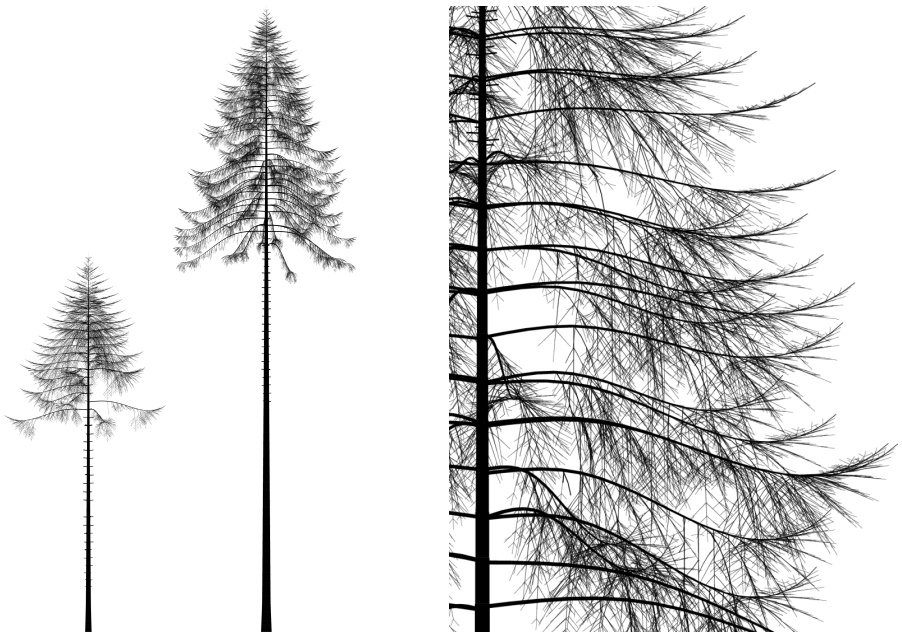


Figure 3.8. Outcome of spruce model at the age of 71 and 113 years, respectively, and a magnified extract

The software comes with a lot of environmental programs with which an open L-system can communicate. The communication is implemented by a binary protocol which is transferred through either pipes, sockets, memory, or files.

The set of turtle commands has been considerably enlarged compared to the standard. The turtle commands { and } indicate that the enclosed movements of the turtle shall generate a polygon or a swept surface [149] (called *generalized cylinder* within the software), which is a spline surface whose centre line follows the path of the turtle and whose cross section is specified by a predefined contour. The turtle command ~ draws a predefined Bézier surface [149]. There are turtle commands for spheres and circles. A set of tropisms can be defined, their associated turtle commands implement rotations which let the movement direction of the turtle converge to a specified direction (e. g., phototropism, geotropism).

vlab and L-Studio are integrated modelling environments, meaning that they include source code editing, interactive visual specification of predefined contour curves, surfaces, functions and materials, and finally a 3D visualization of the outcome with interactive navigation. Figure 3.9 on the following page shows a screenshot of an evaluation version of L-Studio 4.0.

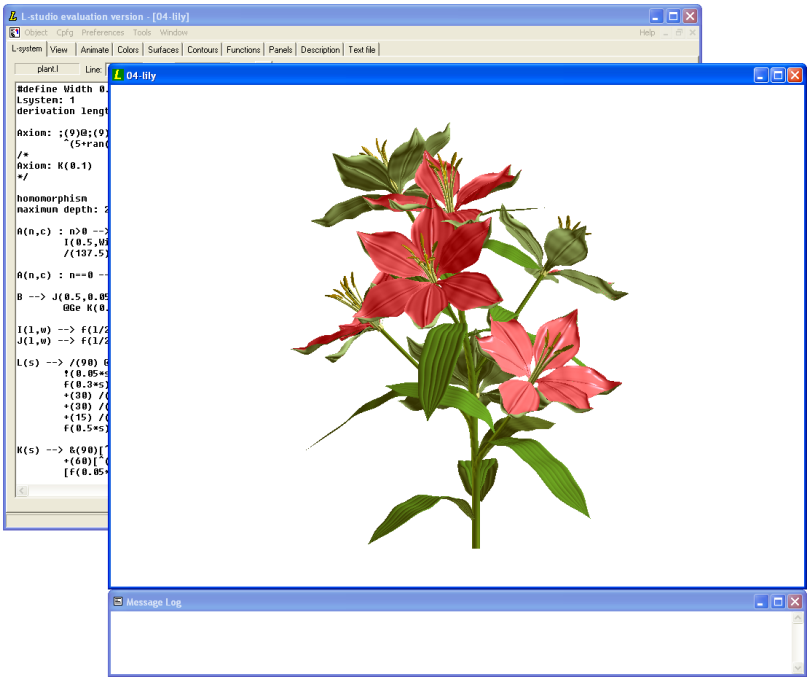


Figure 3.9. Screenshot of the L-Studio software, showing the included example of a lily

3.15.3 Lparser

Lparser [114] is a command-line tool which reads an input file conforming to a subset of the syntax of cpfg and creates a 3D output file which can be displayed by a viewer. For the new version of Lparser [115] the output format is VRML [82] so that it can not only be displayed in a viewer, but also imported into various 3D modellers. A lot of beautiful, artificial images have been created with Lparser, two of which are shown in Fig. 3.10 on the next page.

3.15.4 L-transsys

The program L-transsys [94] differs from the previously presented L-system software in that its principal application is not the execution of L-systems, but the simulation of *gene regulatory networks* by the program component transsys. Within transsys, such a network is a bipartite graph with two types of node elements, transcription factors and genes. Edges from transcription factors to genes represent the activating and repressing effects of factors on the expression of genes, i. e., on the construction of proteins according to the DNA sequences of the genes. Edges from genes to transcription factors (which are proteins) specify which genes encode a transcription factor. Under the

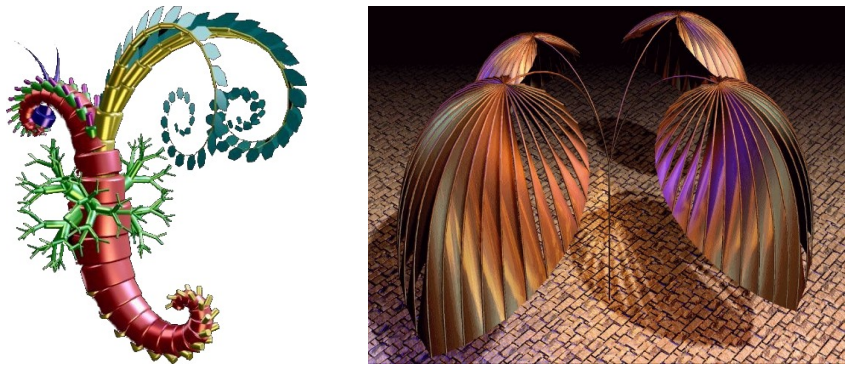
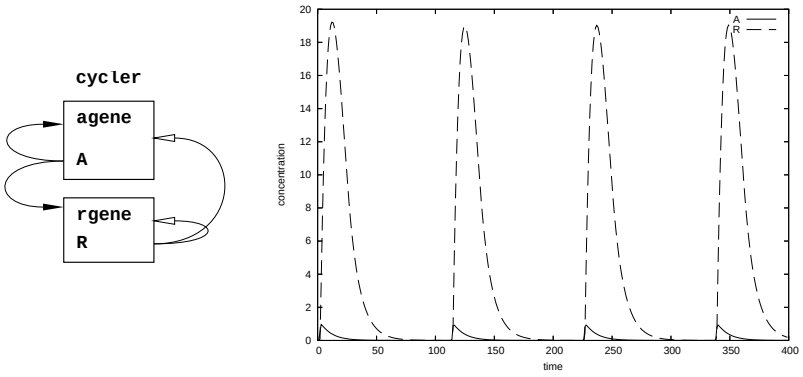


Figure 3.10. Images created with Lparser [114]

simplifying assumption that gene expression and the effect of activation and repression on gene expression can be modelled by enzymatic reactions with Michaelis-Menten kinetics [11], such a network encodes an ordinary differential equation with respect to time for the concentrations of transcription factors.

The transsys program reads a description of the regulatory network as its input and numerically integrates this set of differential equations, the result being a series of factor concentrations. For example, the input

```
transsys cycler {
    factor A { decay: 0.1; }
    factor R { decay: 0.1; }

    gene agene {
        promoter {
            constitutive: 0.01;
            A: activate(0.01, 1.0);
            R: repress(0.1, 1.0);
        }
        product {
            default: A;
        }
    }

    gene rgene {
        promoter {
            A: activate(1.0, 10.0);
            R: repress(1.0, 1.0);
        }
        product {
            default: R;
        }
    }
}
```



(a) (b)

Figure 3.11. Gene regulatory networks in transsys: (**a**) graph representation; (**b**) resulting oscillatory dynamics

specifies the network of Fig. 3.11(a), its resulting oscillatory course of concentrations is depicted in (b).

The extension L-transsys combines this numerical integration of regulatory networks with a variant of parametric L-systems: a parameterized symbol bears a complete set of factor concentrations as parameters, these are automatically updated according to the set of differential equations as part of an interleaved derivation process:

1. Create the axiom with initial factor concentrations.
2. For all symbols parameterized with factor concentrations, compute the new values of factor concentrations after a unit time step.
3. Execute a direct derivation of the L-system. As a parameterized L-system, a derivation may use parameters in conditions and for the successor.
4. Continue with step 2 as long as desired.

L-transsys can thus be seen as an extension of L-systems which incorporates gene regulatory networks as a built-in feature, just like sensitive growth grammars contain a set of globally sensitive functions as built-in features. The following code makes use of the preceding example `cycler` within symbols of type `meristem`.

```
lsys example {
    symbol meristem(cycler);
    symbol shoot_piece;
    symbol left;
    symbol right;
    symbol [;
    symbol ];

    axiom meristem();
```

```
    rule grow { // a single generative production
        meristem(t) : t.A > 0.91 -->
            [ left meristem(transsys t: ) ]
            [ right meristem(transsys t: ) ]
            shoot_piece meristem(transsys t: )
    }

    graphics { // list of interpretive productions
        meristem {
            move(0.2);
            color(0.0, 1.0, 0.0);
            sphere(0.2);
            move(0.2);
        }
        shoot_piece {
            move(0.5);
            color(0.7, 0.7, 0.7);
            cylinder(0.1, 1.0);
            move(0.5);
        }
        left { turn(20); }
        right { turn(-20); }
        [ { push(); }
        ] { pop(); }
    }
}
```

Note that this allows a direct representation of L-system models whose morphogenesis is governed by gene regulatory networks, but both components of the model have to be specified by strictly separated parts of the language. Also there is not yet a feedback from generated structures to regulatory networks, so although we may classify L-transsys models as functional-structural plant models, a flow of information exists only from the functional part to the structural part.

4

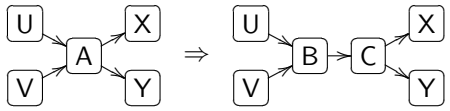
Graph Rewriting

In this chapter, we give an overview of the theory of graph rewriting, and we present some existing tools to apply graph rewriting in practice. This chapter does not contain new results, but it is an important prerequisite for the following chapter.

4.1 Introduction

Graph rewriting or *graph transformation* is the generalization of string rewriting from strings to graphs, i. e., to sets of nodes which are connected by edges. The starting point of graph rewriting was a grammar-based approach for pattern recognition in images [146], but nowadays graph rewriting is used in a variety of fields of application.

While the string-rewriting production $\mathsf{A} \to \mathsf{BC}$, applied to the word UAX, yields the word UBCX, a corresponding graph production $\boxed{\mathsf{A}} \to \boxed{\mathsf{B}} \longrightarrow \boxed{\mathsf{C}}$ could lead to this derivation:



In our notation, directed edges are shown as arrows, while nodes are represented by rounded boxes. The symbols within boxes are *node labels* or *node types*. From this example, three essential questions can be deduced which have to be answered in order to formalize the process of graph rewriting [31]:

1. What is the structure of a graph? In the example, nodes are labelled, while edges are directed but unlabelled. Do we allow parallel edges or loops, i. e., edges from a node to itself?
2. How is the match of the left-hand side of a production at a location in the graph defined? In the example, the A-labelled node of the left-hand side was matched with the A-labelled node of the given graph. This is

similar to string-rewriting formalisms: the symbol of the left-hand side of a production has to match with a symbol in the current word.

3. How do we replace the match by the right-hand side of the production, i. e., what is a direct derivation? Within the context of string rewriting, this is a trivial question: the symbols of the right-hand side simply replace the matched symbol at its location in the word. Within the context of graph rewriting, there is no unique answer to this question. Several so-called *embedding mechanisms* exist which define how a match is removed from the graph and how a right-hand side is attached to the graph. The embedding mechanism used in the example redirects incoming edges of the A-labelled node to the B-labelled node and outgoing edges of the A-labelled node to the C-labelled node. One could equally well choose to redirect all edges from the A-labelled node to, say, the B-labelled node, but then the result would be different.

As we can learn from these questions, the situation is not as straightforward as for string-rewriting systems. Each question permits a large degree of freedom. For the sake of simplicity, we will use the following definition of a labelled, directed graph throughout this chapter [54]:

Definition 4.1 (graph). *Let $\Lambda = (\Lambda_V, \Lambda_E)$ be an alphabet of node labels Λ_V and edge labels Λ_E. A* labelled, directed graph *$G = (G_V, G_E, G_\lambda)$ over Λ consists of a set of* nodes *G_V, a set of* edges *$G_E \subseteq \{(s, \beta, t) | s, t \in G_V, \beta \in \Lambda_E\}$ and a* node labelling function *$G_\lambda : G_V \to \Lambda_V$. For an edge $(s, \beta, t) \in G_E$, s is called the* source node*, t the* target node*. If we use a graph G in a context where it plays the role of a set, e. g., if we write $a \in G$ or $G \subseteq H$, we mean the set $G_V \cup G_E$. A* discrete graph *is a graph without edges, i. e., $G_E = \emptyset$.*

This definition excludes parallel edges of the same label, but allows loops, i. e., edges from a node to itself. It should be clear how this definition can be simplified in order to obtain node-labelled graphs (i. e., edges are unlabelled) or undirected graphs (i. e., the two nodes with which an edge is incident are not distinguished as source and target). In the literature, one also finds definitions which allow parallel edges. Then edges are given by a set and mappings from the set to labels and source and target nodes. Edges are identified by the elements in the set, while an edge of our definition is uniquely determined by its incident nodes and the edge label.

Note that there is a crucial difference in how one conceives of words on the one hand and of graphs on the other hand. Words are usually imagined to be a sequence of *occurrences* of symbols from an alphabet, i. e., while symbols have an identity on their own, a word is built from mere occurrences. On the other hand, nodes of a graph are objects on their own right, they have an identity and are not considered as mere occurrences of their label. Thus, we do not say 'Node A is replaced by ...', which would correspond to 'Symbol A is replaced by ...', but 'A node labelled A is replaced by ...'.

On the basis of the definition of a graph, subgraphs and *graph homomorphisms*, i. e., structure-preserving functions can be defined:

Definition 4.2 (subgraph). *Let $G = (G_V, G_E, G_\lambda)$ be a graph over Λ. A* subgraph *S of G, written $S \sqsubseteq G$, is a graph (S_V, S_E, S_λ) with $S_V \subseteq G_V, S_E \subseteq G_E, S_\lambda = G_\lambda|_{S_V}$.*

Definition 4.3 (graph homomorphism). *Let $G = (G_V, G_E, G_\lambda), H = (H_V, H_E, H_\lambda)$ be graphs over the alphabet Λ and let f_V be a function $G_V \to H_V$. f_V is called* label-preserving *if $H_\lambda \circ f_V = G_\lambda$. f_V induces a function $f_E : G_E \to H_V \times \Lambda_E \times H_V, (s, \beta, t) \mapsto (f_V(s), \beta, f_V(t))$. If $f_E(G_E) \subseteq H_E$ and f_V is label-preserving, f_V induces a* (total) graph homomorphism *$f = (f_V, f_E) : G \to H$. If f is injective, it is called a* monomorphism*. For an object $a \in G_V \cup G_E$, we define $f(a) = f_V(a)$ if $a \in G_V$, otherwise $f(a) = f_E(a)$.*

The labels V and E are omitted from the component functions if it is clear from context which component is meant.

Remark 4.4. If f_V is injective, then f_E is injective. Thus, f is a monomorphism if and only if f_V is injective. Furthermore, f is uniquely determined by f_V. Note that if we did not allow loops as edges, the construction of f_E, given f_V, would exclude functions f_V which send nodes that are connected by an edge to a single node in the image.

With the help of these definitions, we can answer the second question. We have not yet defined exactly what a graph production is, but at the moment it is sufficient to think of a production $L \to R$ where L and R are graphs.

Definition 4.5 (match). *A* match *(i. e., occurrence of the left-hand side) for a production $L \to R$ in some* host graph *G is a graph homomorphism $m : L \to G$.*

Given a host graph G, a production p and a match m, the corresponding derivation should remove the matched part $m(L)$ of the host graph and insert an isomorphic copy of R. How this is done precisely depends on the embedding mechanisms and is discussed in the following section. Given such an embedding mechanism, a set of productions and an initial graph, the definition of a (sequential) *graph grammar* can be stated straightforwardly:

Definition 4.6 (graph grammar). *Let alphabet and embedding mechanism be fixed, i. e., for every production $p : L \to R$ and every (applicable) match $m : L \to G$ a direct derivation $G \overset{p,m}{\Longrightarrow} H$ is defined. A* graph grammar *$\mathcal{G} = (\alpha, P)$ is given by a* start graph *α and a set P of productions. The language generated by $\mathcal{G}$ is $L(\mathcal{G}) = \left\{ G \,\middle|\, \exists \alpha \overset{\mathcal{G}}{\Rightarrow}{}^* G \right\}$, i. e., the set of all graphs G for which there exists a sequence of direct derivations starting at α.*

4.2 Embedding Mechanisms

An embedding mechanism defines how (an isomorphic copy of) the right-hand side R of a production is embedded into the remainder G^- of the host

graph G after the match $m(L)$ of the left-hand side has been removed, i. e., it establishes the link between G^- and R. This mechanism e may depend on the production, so in general, a production has to be written as $L \xrightarrow{e} R$. The embedding mechanism can be seen as the most essential part of graph rewriting; this is in contrast to string rewriting where embedding is trivial.

Two main types of embedding can be distinguished [54]: connecting and gluing. *Connecting* mechanisms explicitly specify new edges which have to be created as bridges between G^- and R. *Gluing* mechanisms establish a link between G^- and R by identifying some objects (nodes, edges) in L and R as part of the production specification. Since these are, roughly spoken, both in the graph L to be deleted and in the graph R to be inserted, their image under m is simply kept in the graph G^-. The remaining part of R is then added to G^-, the link to G^- being established by the kept objects.

We present the most important embedding mechanisms in the sequel. The level of detail varies depending on the importance of the mechanism for the remainder of this thesis.

4.2.1 Neighbourhood Controlled Embedding

Neighbourhood controlled embedding (NCE) is an embedding of the connecting type [54]. A very simple case is the *node label controlled* (NLC) mechanism for node-labelled, undirected graphs. A production has the form $L \xrightarrow{e} R$, where L is a single node, R a graph and e a set of *connection instructions* (μ, ν) with node labels μ, ν. Given a match $m : L \to G$, the single node $m(L)$ in the host graph is removed together with its incident edges. Then for every neighbour a of $m(L)$ in the original host graph and every new node b of (an isomorphic copy of) R, an edge is created if the set of connection instructions contains the pair of labels $(G_\lambda(a), R_\lambda(b))$. Since the left-hand sides are restricted to single nodes, one speaks of *node replacement*.

The basic idea of NLC embedding can be extended easily in various ways, leading to the family of neighbourhood controlled embedding mechanisms. This denomination stresses the locality of embedding: new embedding edges are only created between direct neighbours of the removed nodes on one side and new nodes on the other side. In the case of labelled, directed graphs, we have the edNCE mechanism (e stands for edge-labelled, d for directed):

Definition 4.7 (edNCE production). *An* edNCE production $p : L \xrightarrow{e} R$ *is given by two graphs* L, R *and a finite set* $e = \{c_1, \ldots, c_n\}$ *of* connection instructions $c_i = (v, \mu, \gamma/\delta, w, d) \in L_V \times \Lambda_V \times \Lambda_E \times \Lambda_E \times R_V \times \{\text{in}, \text{out}\}$.

A connection instruction $(v, \mu, \gamma/\delta, w, \text{out})$ specifies that each γ-labelled edge from the removed node v to a μ-labelled neighbour n leads to a δ-labelled embedding edge from the new node w to n, and analogously for 'in' instead of 'out' (d indicates the direction of edges, either incoming or outgoing with respect to the removed node). Note that this embedding is able to relabel

edges, but cannot invert their direction. The mechanism is specified formally by the following definition:

Definition 4.8 (direct edNCE derivation). *Let p be as before, G a labelled, directed graph, $m : L \to G$ a match for p. The* direct derivation *using p via m, denoted as $G \overset{p,m}{\Longrightarrow} H$, is given by the graph H with*

$$
\begin{aligned}
H_V &= (G_V \setminus m(L_V)) \cup R'_V, \\
H_E &= \{(s,\beta,t) \in G_E \,|\, s,t \notin m(L_V)\} \cup R'_E \\
&\quad \cup \bigcup_{(v,\mu,\gamma/\delta,w',d)\in e, d=\text{out}} \{(w',\delta,n) \,|\, \exists (m(v),\gamma,n) \in G_E : G_\lambda(n) = \mu, n \notin m(L_V)\} \\
&\quad \cup \bigcup_{(v,\mu,\gamma/\delta,w',d)\in e, d=\text{in}} \{(n,\delta,w') \,|\, \exists (n,\gamma,m(v)) \in G_E : G_\lambda(n) = \mu, n \notin m(L_V)\}, \\
H_\lambda(v) &= \begin{cases} G_\lambda(v) & : \quad v \in G_V \\ R'_\lambda(v) & : \quad v \in R'_V \end{cases}.
\end{aligned}
$$

R' denotes an isomorphic copy of R.

The introductory example on page 43 could be implemented by this edNCE production, where we write $*$ for the single edge label:

$$
\begin{aligned}
p &= {}^a\boxed{\mathsf{A}} \overset{e}{\to} {}^b\boxed{\mathsf{B}} \longrightarrow {}^c\boxed{\mathsf{C}}, \\
e &= \{(a,\{\mathsf{U},\mathsf{V}\},*/*,b,\text{in}),(a,\{\mathsf{X},\mathsf{Y}\},*/*,c,\text{out})\}\ .
\end{aligned}
$$

In this notation, lowercase letters in front of node boxes are used as node identifiers: for example, the right-hand side R of p is given by $R_V = \{b,c\}, R_E = \{(b,*,c)\}, R_\lambda : b \mapsto \mathsf{B}, c \mapsto \mathsf{C}$. In addition, a set of labels like $\{\mathsf{U},\mathsf{V}\}$ in a connection instruction has to be interpreted as a shorthand notation for a set of connection instructions which use the elements of the set.

4.2.2 Hyperedge Replacement

Hyperedge replacement is an elementary approach of graph and hypergraph rewriting [38]. The underlying graph model is not that of Def. 4.1 on page 44, a (hyper-)graph is rather consisting of a number of *hyperedges* and nodes. Each hyperedge has a fixed number of tentacles, each tentacle is connected with tentacles of other hyperedges at an *attachment node*. In fact, nodes simply serve to collect the attached tentacles of hyperedges and have no further meaning.

As the name indicates, hyperedge replacement replaces hyperedges and keeps nodes unaffected. Left-hand sides of productions are restricted to single hyperedges. On derivation, a matching hyperedge e for the left-hand side is removed, the replacing structure R is embedded into the original structure. In order for this to be possible, R has to be a hypergraph with as many *external nodes* as e has tentacles, these external nodes are then identified with the

previous attachment nodes of e. The embedding of hyperedge replacement is thus of gluing type.

This replacement is context-free in the sense that the context of a replaced hyperedge is not affected. Note that this is the same for string grammars, with symbols instead of hyperedges and the implicit "space between consecutive symbols" instead of attachment nodes. Hyperedge replacement can thus be seen as a direct generalization of string grammars, and several structural results for string grammars can be restated for hyperedge grammars.

4.2.3 Double-Pushout Approach

The *double-pushout approach* (DPO) is an embedding of the gluing type [51, 48, 31, 44]. It is also called the algebraic approach because it relies on an algebraic construction based on category theory. The embedding is specified by a *gluing graph* K which is the common part of the left- and right-hand sides L and R of a production and, thus, identifies objects in R with objects in L. Hence, a rule is specified by $p : L \xrightarrow{K} R$, which is more suitably written as $p : L \leftarrow K \rightarrow R$ in this context:

Definition 4.9 (DPO production). *A* DPO production $p : L \xleftarrow{l} K \xrightarrow{r} R$ *is given by graphs L, K, R, called the left-hand side, gluing graph, right-hand side respectively, and monomorphisms l, r.*

Without loss of generality, we may think of K as the intersection graph of L and R, then l and r are simply inclusions and we write $p : L \hookleftarrow K \hookrightarrow R$.

In order to proceed, we need some basic definitions and constructions from category theory [111]:

Definition 4.10 (category). *A (small)* category *is given by a set of* objects Obj*, for every ordered tuple (A, B) of objects a set of* arrows *(also called* morphisms*)* $\mathrm{Hom}(A, B)$*, where $f \in \mathrm{Hom}(A, B)$ is written as $f : A \rightarrow B$ or $A \xrightarrow{f} B$, for every ordered triple (A, B, C) of objects a* composition *function* $\mathrm{Hom}(B, C) \times \mathrm{Hom}(A, B) \rightarrow \mathrm{Hom}(A, C)$*, written as $(g, f) \mapsto g \circ f$, and for each object A an* identity arrow $\mathrm{id}_A \in \mathrm{Hom}(A, A)$*, such that the following axioms are satisfied:*

1. Associativity. *Given objects and arrows as in $A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D$, one always has the equality $h \circ (g \circ f) = (h \circ g) \circ f$.*
2. Identity. *For all arrows $A \xrightarrow{f} B$, it holds that $f \circ \mathrm{id}_A = \mathrm{id}_B \circ f = f$.*

For an arrow $A \xrightarrow{f} B$, $A = \mathrm{dom}\, f$ is called its domain*, $B = \mathrm{cod}\, f$ its* codomain.

The standard example for a category is **Set**, the category of all small sets.[1] The objects of **Set** are all small sets, $\mathrm{Hom}(A, B)$ is the set of all functions

[1] For the definition of small, see [111].

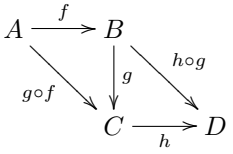
$f : A \to B$, $\circ$ and id are defined as usual. Note that an arrow $A \xrightarrow{f} B$ has its domain A and codomain B as intrinsic properties. E. g., in **Set** the arrows $\mathbb{R} \xrightarrow{x \mapsto x^2} \mathbb{R}$ and $\mathbb{R} \xrightarrow{x \mapsto x^2} \mathbb{R}_0^+$ are different.

Based on **Set**, sets with a structure and their homomorphisms (i. e., structure-preserving functions) define categories, for example the category **Group** of all groups and their homomorphisms, or the category **Graph** which will play an important role in the following:

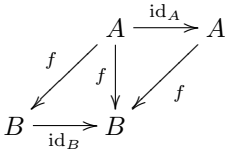
Proposition 4.11 (category Graph). *Graphs and their graph homomorphisms define a category* **Graph**. □

The philosophy behind categories is to describe properties of mathematical systems by a convenient presentation with diagrams consisting of arrows. This notation provides a unified and simplified view at a high level of abstraction. One is not interested in the concrete meaning of an object $A \in \mathrm{Obj}$. Although this may be a set of elements $x \in A$, one never uses such an element x: reasoning exclusively uses arrows. This high level of abstraction is jokingly, but not in the least derogatorily termed *abstract nonsense* [112, page 759]. As an example for the diagrammatic language of category theory, its axioms can be depicted as *commutative diagrams*:

Associativity :

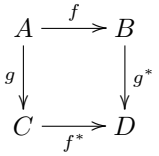


Identity :

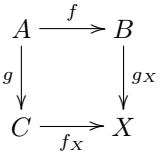


Commutative diagrams have to be read as follows: for each pair of objects X, Y in such a diagram and for any two paths $X \xrightarrow{f_1} \ldots \xrightarrow{f_n} Y$, $X \xrightarrow{g_1} \ldots \xrightarrow{g_m} Y$ along arrows in their direction, the equality $f_n \circ \ldots \circ f_1 = g_m \circ \ldots \circ g_1$ holds.

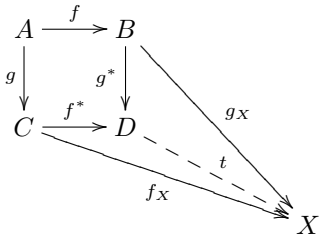
Definition 4.12 (pushout). *Let $B \xleftarrow{f} A \xrightarrow{g} C$ be two arrows in a category. A* pushout (D, f^*, g^*) *over f, g is a commutative square*



which satisfies the following universal property*: for any commutative square*



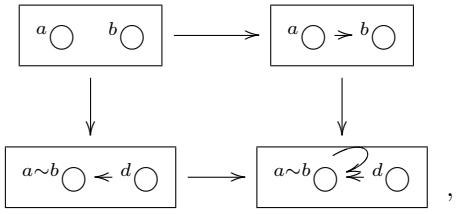
there exists a unique arrow $D \xrightarrow{t} X$ such that the following diagram commutes:



By this definition, the pushout is unique up to isomorphism. In the category **Set**, such a pushout always exists [111]: it is given by $(B \sqcup C)/\sim^*$, the disjoint union of B and C modulo the least equivalence relation $\sim^*$ which identifies $f(a)$ and $g(a)$ for each $a \in A$. This gives us an intuitive meaning: if f, g are inclusions, D consists of the elements of A plus those of B and C which are not already in A. The pushout glues B and C on common elements. This construction can be adopted to **Graph**:

Proposition 4.13 (pushouts in Graph). *Pushouts in* **Graph** *always exists. They can be obtained by the pushout construction of* **Set** *with respect to the node component, which then uniquely determines the edge component.* □

Note that this construction relies on the fact that loops are allowed as edges. An example for a pushout in **Graph** is the diagram



where node mappings are indicated by lowercase identifiers. $a \sim b$ as identifier means that both a and b are mapped to the same node.

Given a host graph G and a match $m : L \to G$, the corresponding direct derivation can now be defined in pure categorical terms:

Definition 4.14 (direct DPO derivation). *Let $p : L \hookleftarrow K \hookrightarrow R$ be a production and $m : L \to G$ a match in a host graph G. A* direct derivation *using p via m, denoted as $G \overset{p,m}{\Longrightarrow} H$, is given by the following double-pushout diagram, where* (1) *and* (2) *are pushouts in the category* **Graph***:*

$$\begin{array}{ccccc} L & \hookleftarrow & K & \hookrightarrow & R \\ {\scriptstyle m}\downarrow & (1) & {\scriptstyle d}\downarrow & (2) & \downarrow{\scriptstyle m^*} \\ G & \underset{l^*}{\leftarrow} & D & \underset{r^*}{\rightarrow} & H \end{array}$$

D is called the context graph*, m^* the* comatch*.*

If D and H exist, they are uniquely determined up to isomorphism by this diagram. This is a common characteristic of the algebraic approach to graph rewriting and reasoning in category theory in general: objects are determined up to isomorphism only. This can be seen as an abstraction of representation-dependent details. From a practical point of view, it is clear how to deal with this issue: wherever a diagram such as a pushout determines objects up to isomorphism only, one simply fixes a specific construction for this diagram which is then unique. A thorough investigation from a theoretical point of view is presented in [29, 30], where suitable equivalence relations and standard representations are studied.

It is not always possible to construct the double-pushout diagram: given pushout (1), H can always be constructed as pushout (2), but pushout (1) requires its completion by D which is only possible if and only if the *gluing condition* is fulfilled:

Definition 4.15 (applicability of production). *Let G, p, m be as before. p is* applicable *via m if the double-pushout diagram exists.*

Definition 4.16 (gluing condition). *Let G, p, m be as before. The* identification points $IP = \{a \in L \mid \exists b \in L, a \neq b : m(a) = m(b)\}$ *are those nodes and edges in L which are identified by m. The* dangling points $DP = \{v \in L_V \mid \exists (s, \beta, t) \in G_E \setminus m(L_E) : s = m(v) \vee t = m(v)\}$ *are those nodes in L whose images under m are incident with edges that do not belong to $m(L)$. The* gluing condition *is $IP \cup DP \subseteq K$.*

The gluing condition says that nodes of L which are identified by a match must not be deleted, and that nodes which are removed by a derivation must not leave dangling edges behind.

Proposition 4.17 (existence and uniqueness of context graph). *Let G, p, m be as before. The context graph D for a direct derivation as in Def. 4.14 on the facing page exists if and only if the gluing condition is fulfilled. If D exists, it is unique up to isomorphism, and the double-pushout diagram can be constructed, i. e., the production p is applicable.* [44] □

The concrete implementation of this construction is as follows [44]. Given a derivation using $p : L \hookleftarrow K \hookrightarrow R$ via $m : L \to G$, delete those nodes and edges in G that are reached by the match m, but keep the image of K. This results in the context graph $D = (G \setminus m(L)) \cup m(K)$. Then add those nodes and edges that are newly created in R. This results in $H = D \sqcup (R \setminus K)$. We use the disjoint union to express that the added elements are really new.

As an example for a DPO derivation, consider Fig. 4.1 on the next page. m is not injective, it identifies a and b in G. However, since both are elements of K, this does not violate the gluing condition. Also a, b are dangling points because their image is connected with d. But again this does not violate the gluing condition since $a, b \in K$. Thus, the production is applicable and amounts to the deletion of c and the creation of a loop edge at $a \sim b$.

The two examples in Fig. 4.2 show situations where the gluing condition is not satisfied and, hence, a DPO derivation cannot be constructed. Even the simple production of Fig. 4.2(b) which just deletes a node cannot be applied to a node in G which has edges.

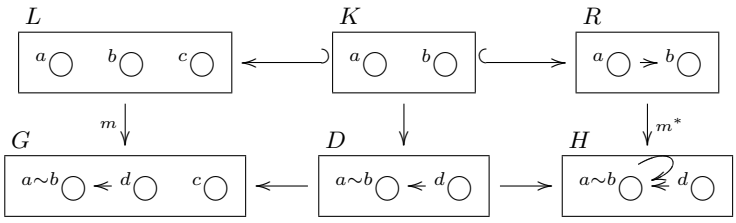


Figure 4.1. DPO derivation

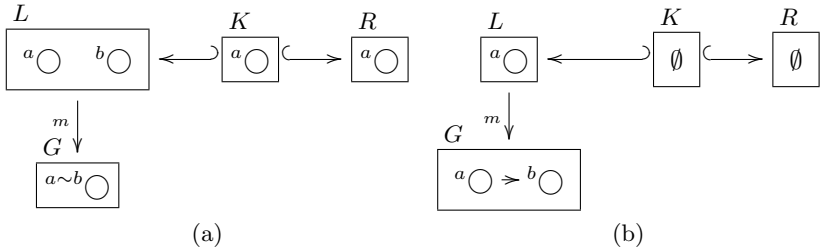


Figure 4.2. Inapplicable productions: (**a**) identification point $b \notin K$; (**b**) dangling point $a \notin K$

4.2.4 Single-Pushout Approach

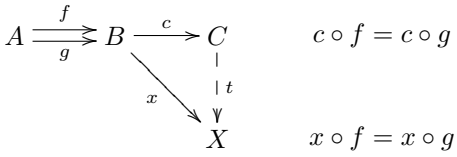
Like the double-pushout approach, the *single-pushout approach* (SPO) is an embedding of the gluing type [48, 47]. It has been developed as a generalization of the DPO approach by dispensing with the gluing condition. The conflicts which are indicated by a violated gluing condition are resolved by deletion, i. e., dangling edges and objects which shall both be preserved and deleted are deleted. The fact that a production is applicable via every match is a consequence of the SPO construction where a derivation is a single pushout diagram which always exists. However, we have to change the underlying category from **Graph** to $\mathbf{Graph_P}$ where arrows are partial graph homomorphisms:

Definition 4.18 (partial graph homomorphism). *Let G, H be graphs over the alphabet Λ. A* partial graph homomorphism *f from G to H is a total graph homomorphism from some subgraph* $\mathrm{dom}_P\, f$ *of G to H, and* $\mathrm{dom}_P\, f$ *is called the* domain *of f.* [2]

[2] Note that the term *domain* is used ambiguously here: $\mathrm{dom}_P\, f \sqsubseteq G$ is the domain of $G \xrightarrow{f} H$ as a partial function, $G = \mathrm{dom}\, f$ is its domain as an arrow (see Def. 4.10 on page 48).

Proposition 4.19 (category $\mathbf{Graph_P}$). *Graphs and their partial graph homomorphisms define a category* $\mathbf{Graph_P}$. □

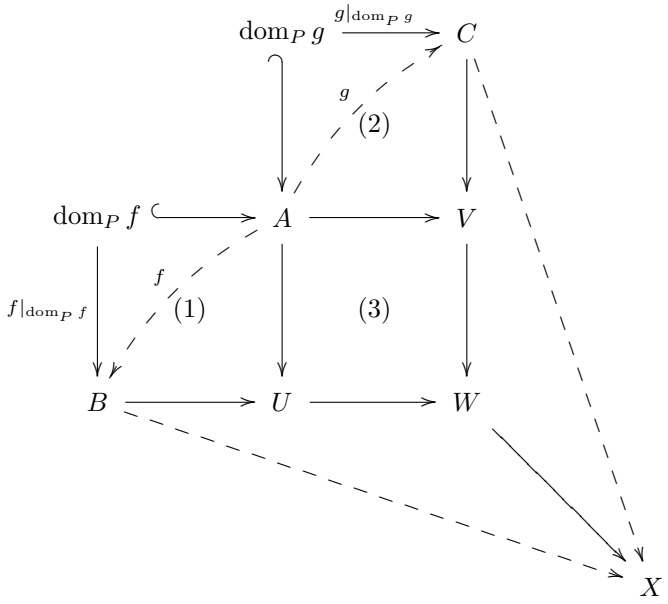
Definition 4.20 (coequalizer). *Let $f, g : A \to B$ be two arrows. A* coequalizer *of f, g is an arrow $B \xrightarrow{c} C$ such that $c \circ f = c \circ g$ and for any arrow $B \xrightarrow{x} X$ with $x \circ f = x \circ g$ there exists a unique arrow $C \xrightarrow{t} X$ with $x = t \circ c$:*



Coequalizers are unique up to isomorphism. Note that only the right part of the coequalizer diagram is commutative.

Proposition 4.21 (specific coequalizer in $\mathbf{Graph_P}$). *Let $f, g : A \to B$ be two partial graph homomorphisms which coincide on $\mathrm{dom}_P\, f \cap \mathrm{dom}_P\, g$. Then a coequalizer always exists and is given by the partial graph morphism $B \xrightarrow{c} C$ where $C \sqsubseteq B$ is the largest subgraph of B with $C \subseteq \{a \in B \mid f^{-1}(a) = g^{-1}(a)\}$, $\mathrm{dom}_P\, c = C$, and c is the identity on C.* [122] [3] □

Proposition 4.22 (construction of pushouts in $\mathbf{Graph_P}$). *Pushouts for partial graph homomorphisms $B \xleftarrow{f} A \xrightarrow{g} C$ always exist and can be constructed by the following diagram:*



[3] The original construction in [47] is wrong and was corrected in [122].

At first, the pushouts (1) *of* $A \hookleftarrow \mathrm{dom}_P f \xrightarrow{f|_{\mathrm{dom}_P f}} B$ *and* (2) *of* $A \hookleftarrow \mathrm{dom}_P g \xrightarrow{g|_{\mathrm{dom}_P g}} C$ *are constructed in* **Graph** *(all arrows are total graph morphisms). Then the pushout* (3) *of* $U \leftarrow A \rightarrow V$ *is constructed in* **Graph**. *Finally, the coequalizer* X *of* $A \xrightarrow{f} B \rightarrow U \rightarrow W$ *and* $A \xrightarrow{g} C \rightarrow V \rightarrow W$ *is constructed in* $\mathbf{Graph_P}$ *(the conditions for the specific construction are fulfilled). The dashed square is the pushout of* $B \xleftarrow{f} A \xrightarrow{g} C$ *in* $\mathbf{Graph_P}$, X *its pushout object.* □

Remark 4.23. Both the solid and dashed part of the diagram are commutative in isolation. However note that the diagram as a whole is not commutative since f, g are only partial graph homomorphisms. It is commutative on a subdomain (start with replacing A by $\mathrm{dom}_P f \cap \mathrm{dom}_P g$).

Remark 4.24. This construction differs from the one in [47]. However, the latter is equivalent and can be obtained by the fact that pushouts can be composed and decomposed, i. e., (2) + (3) is also a pushout and could be constructed at once after (1). The presented construction was chosen instead of the abbreviated one due to its symmetry.

Definition 4.25 (SPO production). *An* SPO production $p : L \rightarrow R$ *is a partial graph monomorphism* $L \xrightarrow{p} R$.

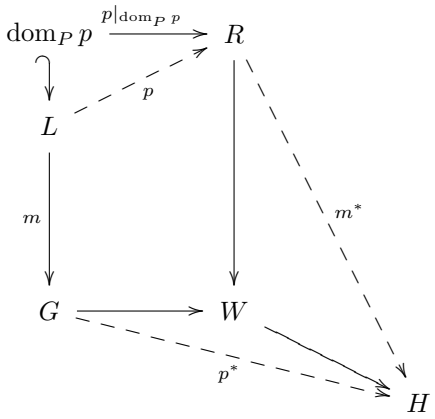
Without loss of generality, we may assume that p is the identity on its domain. We can then rewrite p as $L \hookleftarrow \mathrm{dom}_P p \hookrightarrow R$ which shows that the role of $\mathrm{dom}_P p$ coincides with that of the gluing graph K of a DPO rule.

Definition 4.26 (direct SPO derivation). *Let* $p : L \rightarrow R$ *be a production and* $m : L \rightarrow G$ *a match in a host graph* G. *A* direct derivation *using* p *via* m, *denoted as* $G \overset{p,m}{\Longrightarrow} H$, *is given by the following single-pushout diagram in the category* $\mathbf{Graph_P}$*:*

$$\begin{array}{ccc} L & \xrightarrow{p} & R \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m^*} \\ G & \xrightarrow[p^*]{} & H \end{array}$$

m^* *is called the* comatch.

Remark 4.27. The construction of Prop. 4.22 on the preceding page can be simplified in case of the pushout of an SPO derivation since m is a total graph homomorphism. In the terms of the construction, we have $\mathrm{dom}_P f = A, U = B$ and construct the composed pushout (2) + (3) at once:



The concrete implementation of an SPO derivation starts with the construction of W. Assuming $p|_{\mathrm{dom}_P\, p} = \mathrm{id}_{\mathrm{dom}_P\, p}$ without loss of generality, this amounts to the addition of the new objects $R \backslash \mathrm{dom}_P\, p$ to G, thereby respecting the (transitive closure of the) identification $m(a) \sim a$ for $a \in \mathrm{dom}_P\, p$. Then the coequalizer H is constructed by deletion of all objects of the old host graph G whose preimage under m contains elements of $L \backslash \mathrm{dom}_P\, p$ and, finally, by deletion of all dangling edges. Thus, the apparent difference to a DPO derivation is that the latter at first deletes objects by completing its pushout (1) with the context graph D and then adds new objects, while an SPO derivation at first adds new objects by constructing its pushout object W and then deletes objects. This order implements deletion in preference to preservation.

The SPO derivation of the production of Fig. 4.1 on page 52 looks the same as the DPO derivation. However, the productions of 4.2 are now both applicable as shown in Fig. 4.3.

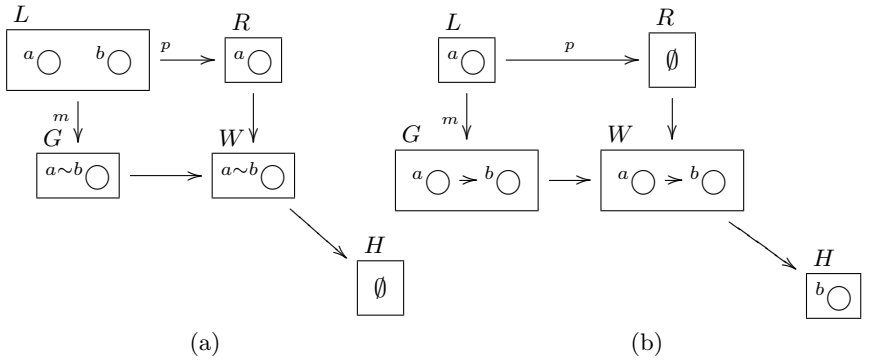


Figure 4.3. Productions with violated gluing condition are applicable in the SPO approach

In [47], the SPO approach is compared with the DPO approach. The DPO approach is considered to be safe because there is no implicit effect of deletion of dangling edges; in fact, DPO derivations are invertible straightforwardly. On the other hand, the SPO approach has the side effect of deletion of dangling edges. SPO derivations are thus not invertible and are considered as relatively unsafe. A way to moderate this unsafety is the inclusion of application conditions. Indeed, the SPO approach with the gluing condition as application condition is equivalent to the DPO approach.

The suggestion of [47] for concrete applications of graph grammars is to assume a set of standard application conditions for productions in order to restrict the expressiveness to a safe and intelligible amount. Exceptions should be possible by explicit specification of the user. Standard application conditions could be the gluing condition or the requirement of injectivity for matches.

4.2.5 Pullback Rewriting

The basic operation of the two presented pushout approaches is the pushout in a suitable category of graphs. As has been noted after the pushout definition 4.12 on page 49, a pushout can be seen as a generalization of a union which glues its components on common elements. This complies with the usual rewriting mechanism of grammars where, after the matched part of a rule application has been removed from the current structure, the new structure is obtained by the union with the right-hand side of the rule and an additional identification of objects.

However, a more powerful rewriting mechanism can be defined based on products rather than on unions [8]. In terms of category theory, products and generalizations thereof can be defined via a *pullback*, the dual of a pushout (i. e., directions of arrows are reversed):

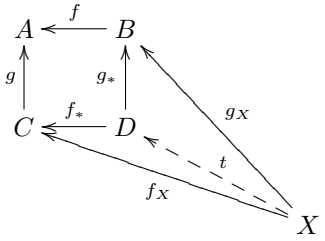
Definition 4.28 (pullback). *Let $B \xrightarrow{f} A \xleftarrow{g} C$ be two arrows in a category. A* pullback (D, f_*, g_*) *over f, g is a commutative square*

$$\begin{array}{ccc} A & \xleftarrow{f} & B \\ {\scriptstyle g}\uparrow & & \uparrow{\scriptstyle g_*} \\ C & \xleftarrow[f_*]{} & D \end{array}$$

which satisfies the following universal property*: for any commutative square*

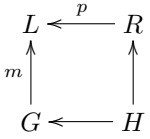
$$\begin{array}{ccc} A & \xleftarrow{f} & B \\ {\scriptstyle g}\uparrow & & \uparrow{\scriptstyle g_X} \\ C & \xleftarrow[f_X]{} & X \end{array}$$

there exists a unique arrow $X \xrightarrow{t} D$ *such that the following diagram commutes:*



By this definition, the pullback is unique up to isomorphism. In the category **Set**, such a pullback always exists [44]: it is constructed by $D = \{(b,c) \in B \times C \,|\, f(b) = g(c)\}$ with homomorphisms $f_* : D \to C, (b,c) \mapsto c$ and $g_* : D \to B, (b,c) \mapsto b$. If A is a singleton set, D is simply the cartesian product $B \times C$. If f and g are inclusions, D is (isomorphic to) the intersection $B \cap C$.

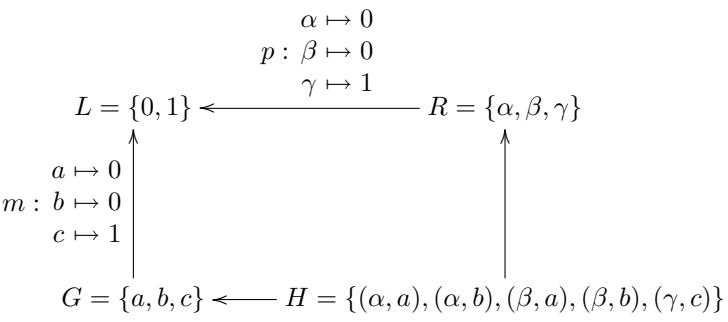
A precise description of pullback rewriting is given in [8, 9]. The following just serves to give an idea of the formalism and its benefits. Since a pullback is the dual of a pushout, pullback rewriting is dual to pushout rewriting: directions of arrows have to be reversed. In a single-pullback framework, a production is thus given by an arrow $p : R \to L$ and a match by an arrow $m : G \to L$. A derivation using p via m is the pullback of $G \xrightarrow{m} L \xleftarrow{p} R$:



Thus, a match assigns to each object of the host graph an object of the left-hand side of a rule. This allows a single object of the left-hand side to be matched by several objects of the host graph.

A simple application of this feature is the implementation of the edNCE mechanism using pullback rewriting [8]. Connection instructions can be encoded as part of productions; namely, for a node v of the left-hand side of a production, the unspecified number of neighbours of the match of v matches a single neighbour $n \in L$ of v. The connection instruction part of the production p specifies how edges from v to n are transferred to the derived graph, this is then carried out for each matching node for n.

A further application is the duplication of whole subgraphs: if a subgraph S of the host graph matches a single node v (with loop) of the left-hand side, and if the homomorphism p maps several nodes of R to v, the pullback creates a copy of S in the derived graph H for every such node. Compare this with the situation



in **Set**: the derived set H is given by $\{(\alpha, a), (\alpha, b), (\beta, a), (\beta, b), (\gamma, c)\}$, i. e., the subset $S = \{a, b\}$ which matches the element 0 of L is duplicated (once tagged with α and once tagged with β) because the two elements α and β of R are mapped to 0 by p, the rest of G matches the element 1 of L and is preserved because only γ is mapped to 1 by p.

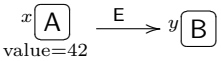
4.2.6 Relation-Algebraic Rewriting

Pushout and pullback rewriting are complementary. As a consequence of the direction of arrows which represent homomorphisms, pushout rewriting may identify several objects of the left-hand side of a production with the same object in the host graph, which is a useful feature for parallel derivations (see remark 4.51 on page 70), but may not identify several objects in the host graph with the same object of the left-hand side so that whole subgraphs cannot be duplicated. On the other hand, pullback rewriting may duplicate subgraphs, but cannot identify objects of the left-hand side.

A unification of these complementary approaches can be obtained if the meaning of an arrow as a (partial) homomorphism is dispensed with in favour of a general relation [88]. A partial homomorphism is then simply the special case of a univalent relation, a total homomorphism is obtained if the relation is also total. In [88] it is argued that this concept is as intuitive and elegant as the pushout approaches while incorporating the replicative power of the pullback approaches (which by themselves are less intuitive).

4.2.7 Logic-Based Structure Replacement Systems

Logic-based structure replacement systems [173] differ from the previous approaches in that they use a completely different graph model. A graph is represented as a so-called Σ-structure which is a set of closed atomic formulas. For example, the graph



would be represented as the set of atomic formulas

$$\{\mathrm{node}(x, \mathsf{A}), \mathrm{node}(y, \mathsf{B}), \mathrm{edge}(x, \mathsf{E}, y), \mathrm{attr}(x, \mathrm{value}, 42)\} \ .$$

Using first-order predicate logic formulas, a Σ-structure schema consisting of integrity constraints and derived data definitions can be specified. The integrity constraint that each edge is incident with nodes is the formula

$$\forall s, \beta, t : \mathrm{edge}(s, \beta, t) \Rightarrow (\exists X, Y : \mathrm{node}(s, X) \wedge \mathrm{node}(t, Y)) \ ,$$

and the derived definition of an ancestor is the formula

$$\forall x, y : \mathrm{ancestor}(x, y) \Leftrightarrow (\exists z : \mathrm{edge}(z, \mathrm{child}, x) \wedge (z = y \vee \mathrm{ancestor}(z, y))) \ .$$

A graph G (or a Σ-structure in general) conforms to a schema if G is consistent with the set Φ of integrity constraints and derived data definitions of the schema, i. e., if $G \cup \Phi$ is free of contradictions.

A match is represented by a Σ-relation which relates the left-hand side of a production with the host structure. The left-hand side may contain object identifiers and object set identifiers as variables. Object identifiers are related with single objects of the host structure, while object set identifiers are related with an arbitrarily large set of objects of the host structure. This is similar to the combination of pushout and pullback rewriting in the relation-algebraic rewriting of Sect. 4.2.6 on the preceding page. As an additional requirement, preconditions have to be fulfilled, and a match for a production has to be maximal, i. e., the matches for object set identifiers must not be extensible.

A derivation at first removes the image of the left-hand side without the image of the right-hand side from the host structure and then adds an image of the right-hand side without the image of the left-hand side to the host structure. As a consequence of this construction, conflicts between deleting and preserving objects are resolved in favour of preserving. Afterwards, post-conditions and schema consistency is checked. If any violation is detected, the derivation has to be undone.

According to [173], logic-based structure replacement systems are an attempt to close the gap between rule-based modification and logic-based knowledge representation. Structure replacement systems are expected to be a well-defined formalism for the modification of knowledge bases, and logic-based techniques can be used to define constraints and derived properties, and to prove the correctness of modifications.

4.3 Parallel Graph Rewriting

The discussion so far deals with sequential rewriting exclusively. But from the early days of graph rewriting on also parallel rewriting was studied, mainly inspired by the contemporaneous evolution of the theory of L-systems [118]. Partly motivated by biological applications, a heterogeneous range of approaches for parallel graph grammars was proposed in the mid and late 1970s

[32, 127, 143, 130, 123, 119], and some efforts were made to generalize and unify them [52, 49]. An early review can be found in [132]. However, at least to our knowledge, none of these approaches is still being an active field of research or used for the modelling of biological systems. In fact, although parallel graph rewriting made it into the early textbook *Graph-Grammatiken* [131] of 1979, they were not discussed in the three-volume *Handbook of Graph Grammars* [167, 45, 50] of 1997.

The main problem of parallel graph rewriting is the extension of the sequential embedding mechanism to a *connection mechanism* suitable for the parallel mode of rewriting. Such a connection mechanism has to state how the right-hand side of a rule has to be connected with the rest, but the rest itself changes as part of a derivation. Two different ways of defining connection mechanisms were proposed, namely *explicit* and *implicit* approaches [132]. Explicit approaches have two kinds of productions: rewriting productions replace predecessor subgraphs of the host graph by successor subgraphs, connection productions define how connections between two predecessor subgraphs are transformed into connections between the corresponding successor subgraphs. Implicit approaches resemble embedding mechanisms of sequential rewriting: there is only one sort of productions which specify both predecessor and successor subgraphs and the way how successor subgraphs are connected with the rest (consisting of all the other successor subgraphs). So explicit approaches establish connections based on a pairwise consideration of successor subgraphs, while implicit approaches connect each successor subgraph with the rest.

4.3.1 Explicit Connection Mechanisms

As a representative for explicit connection mechanisms we present the definition of connection productions based on *stencils*. The main idea was introduced in [32] and was further investigated (with some changes in the definition) in [52, 86, 87]:

1. Replace each single node of the host graph individually and simultaneously by a successor graph according to the set of rewriting productions. Thus, left-hand sides of productions are restricted to single nodes (node replacement).
2. For each edge of the host graph, a connection production is chosen based on the edge label and the successor graphs of its incident nodes. This production specifies the edges to be created between the successor graphs.

A precise definition based on [52] is as follows:

Definition 4.29 (rewriting and connection productions based on stencils). *A* rewriting production $p : \mu \to R$ *is given by a node label* $\mu \in \Lambda_V$ *and a graph* R*. A* connection production $c : \beta \to R$ *is given by an edge label* $\beta \in \Lambda_E$ *and a* stencil R*, where a stencil* $R = (S, T, E)$ *consists of a* source graph S*,*

a target graph T *and additional edges* $E \subseteq S_V \times \Lambda_E \times T_V \cup T_V \times \Lambda_E \times S_V$ *between* S *and* T*.*

Definition 4.30 (node replacement parallel graph grammar). *A* node replacement parallel graph grammar $\mathcal{G} = (\alpha, P, C)$ *is given by a* start graph α*, a set* P *of rewriting productions and a set* C *of connection productions.*

Definition 4.31 (direct derivation). *Let* $\mathcal{G} = (\alpha, P, C)$ *be a node replacement parallel graph grammar, and let* G, H *be graphs. Then there is a (direct)* derivation *from* G *to* H *within* $\mathcal{G}$ *if all of the following holds:*

1. *There is, for each node* $v \in G_V$*, a rewriting production* $p^v : \mu^v \to R^v \in P$ *with* $\mu^v = G_\lambda(v)$ *such that* H_V *is isomorphic (as a discrete graph) to* $\bigsqcup_{v \in G_V} R^v_V$*, the isomorphism being given by* $h : \bigsqcup_{v \in G_V} R^v_V \to H_V$*.*
2. *There is, for each edge* $e = (s, \beta, t) \in G_E$*, a connection production* $c^e : \beta^e \to (S^e, T^e, E^e) \in C$ *with* $\beta^e = \beta$ *such that* S^e *is isomorphic to* R^s *and* T^e *is isomorphic to* R^t*, the isomorphisms being given by* $\sigma^e : S^e \sqcup T^e \to R^s \sqcup R^t$ *(these conditions may be changed, e. g., cases like* $S^e = R^s, T^e = R^t$ *or* $S^e \sqsubseteq R^s, T^e \sqsubseteq R^t$ *are also considered in the literature).*
3. *The set of edges of* H *is given by*

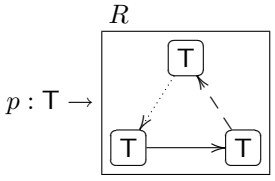
$$\begin{aligned} H_E = & \bigcup_{v \in G_V} \{(h(s), \beta, h(t)) \,|\, (s, \beta, t) \in R^v_E\} \\ & \cup \bigcup_{e \in G_E} \{((h \circ \sigma^e)(s), \beta, (h \circ \sigma^e)(t)) \,|\, (s, \beta, t) \in E^e\} \ . \end{aligned}$$

Roughly speaking, H is the union of all stencils resulting from application of connection productions, glued together in the subgraphs resulting from application of rewriting productions.

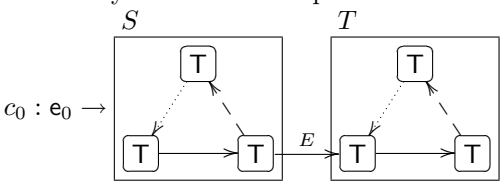
Example 4.32 (Sierpinski grammar). As an example for a node replacement parallel graph grammar let us consider a variant of the Sierpinski construction (see Sect. 2.3 on page 13 and L-system (3.2) on page 21) based on graphs. A node in a graph represents a complete black triangle. So the start graph (the initiator of the Sierpinski construction) is given by

$$\alpha = \boxed{\mathsf{T}} \,,$$

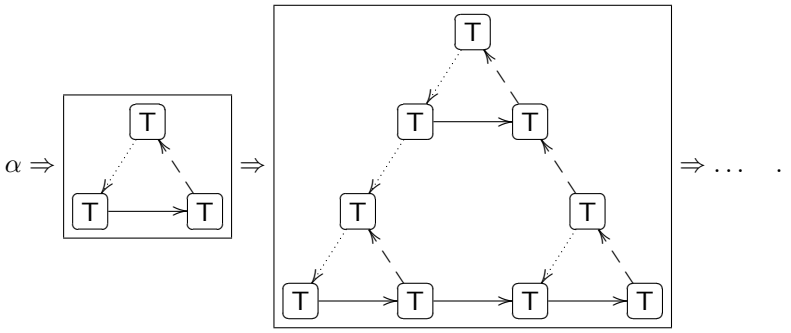
where T is the type of triangle nodes. Now the generator of the Sierpinski construction replaces each black triangle, i. e., each node of type T, by three black triangles. In order to keep track of the topology, we create edges between these new triangles using the edge types e_0, e_{120}, e_{240} which stand for edges at 0°, 120° and 240°, respectively, in the usual 2D representation, and which are drawn as solid, dashed, or dotted arrows. The generator is then given by the rewriting production



Now we also have to transfer the old connecting edges between black triangles to new ones, this is done by the connection production



and the corresponding ones c_{120}, c_{240} for the other edge types. The parallel graph grammar $\mathcal{G} = (\alpha, \{p\}, \{c_0, c_{120}, c_{240}\})$ leads to the sequence of derivations



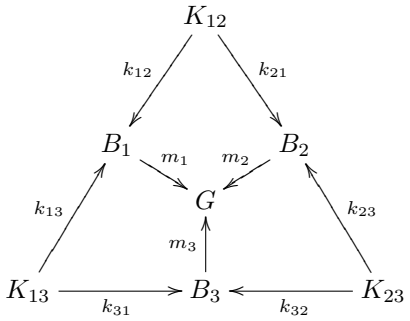
The stencil approach can be reformulated and generalized within the algebraic theory of graph rewriting [49]. Here, we may informally think of a complete covering $G = \bigcup_{i \in I} B_i$ of the host graph by a family of graphs $(B_i)_{i \in I}$ which is the generalization of the covering with stencils. The intersection graphs $K_{ij} = B_i \cap B_j$ (i. e., the largest subgraphs of G contained in the sets $B_i \cap B_j$) generalize the role of nodes, and the parts of B_i which are not contained in some other B_j generalize the role of edges. Now all intersection graphs K_{ij} and all graphs B_i of the covering are replaced by successor graphs K'_{ij}, B'_i such that $K'_{ij} = B'_i \cap B'_j$, these are then glued on the common parts K'_{ij} to yield the derived graph H. The precise definition is the following:

Definition 4.33 (covering grammar production). *A* covering grammar production *is a pair of graph homomorphisms* $p : L \leftarrow K, K' \rightarrow R$. K, K' *may be empty.*

Definition 4.34 (covering parallel graph grammar). *A* covering parallel graph grammar $\mathcal{G} = (\alpha, P)$ *is given by a* start graph α *and a set* P *of covering grammar productions.*

Definition 4.35 (pushout-star). *Let I be a finite index set, $(B_i)_{i\in I}$ a family of objects in a category and $(k_{ij} : K_{ij} \to B_i)_{i\neq j\in I}$ a family of arrows such that $K_{ij} = K_{ji}$. A* pushout-star *is an object G together with arrows $m_i : B_i \to G$ such that $m_i \circ k_{ij} = m_j \circ k_{ji}$ and the universal property is fulfilled (i. e., for any G', m'_i with $m'_i \circ k_{ij} = m'_j \circ k_{ji}$ there is a unique arrow $f : G \to G'$ such that $f \circ m_i = m'_i$).*

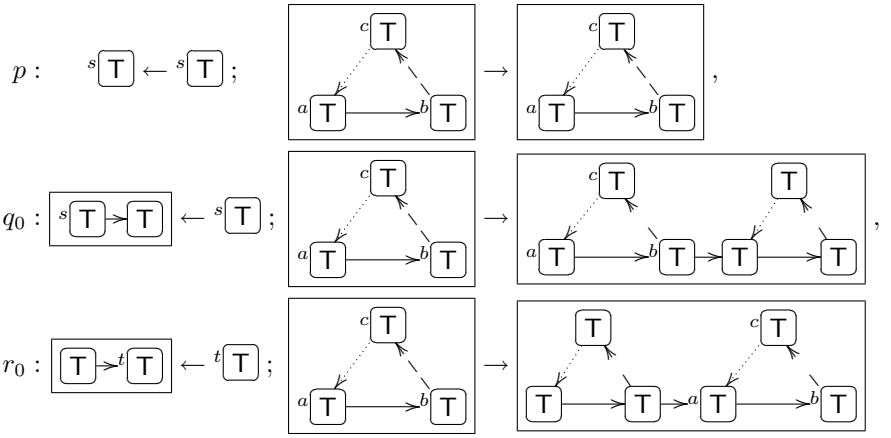
The case $I = \{1, 2, 3\}$ is illustrated by the following diagram:



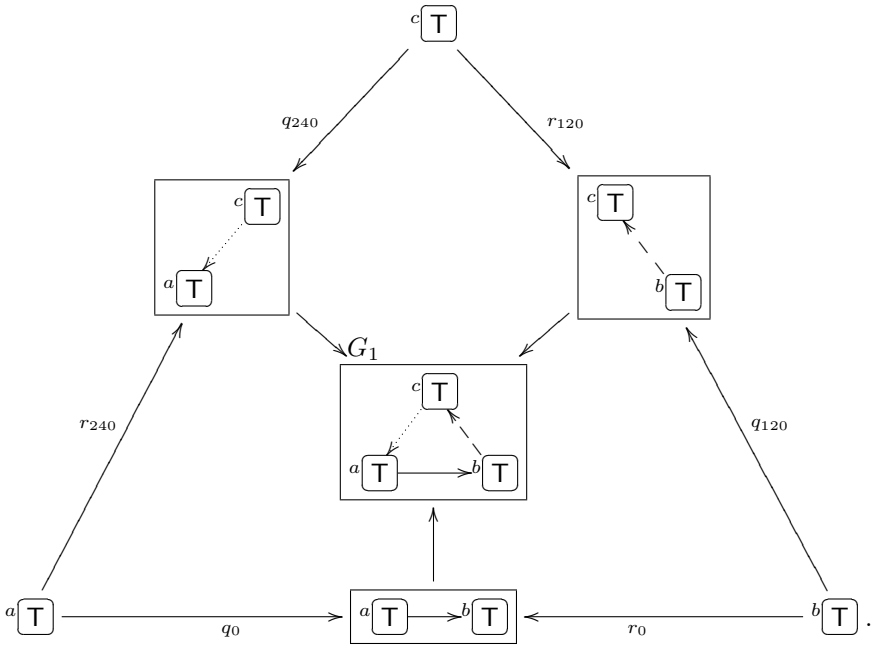
Definition 4.36 (direct derivation). *Let $\mathcal{G} = (\alpha, P)$ be a covering parallel graph grammar, and let G, H be graphs. Then there is a (direct)* derivation *from G to H within $\mathcal{G}$ if all of the following holds:*

1. *There exist a finite index set I and a family $(p_{ij} : L_i \leftarrow K_{ij}; K'_{ij} \to R_i)_{i\neq j\in I}$ of productions $p_{ij} \in P$ with $K_{ij} = K_{ji}, K'_{ij} = K'_{ji}$.*
2. *G is a pushout-star object of $(K_{ij} \to L_i)_{i\neq j\in I}$.*
3. *H is a pushout-star object of $(K'_{ij} \to R_i)_{i\neq j\in I}$.*

Example 4.37 (Sierpinski grammar). The previous node replacement Sierpinski grammar can be reformulated as a covering parallel graph grammar. The productions are given by

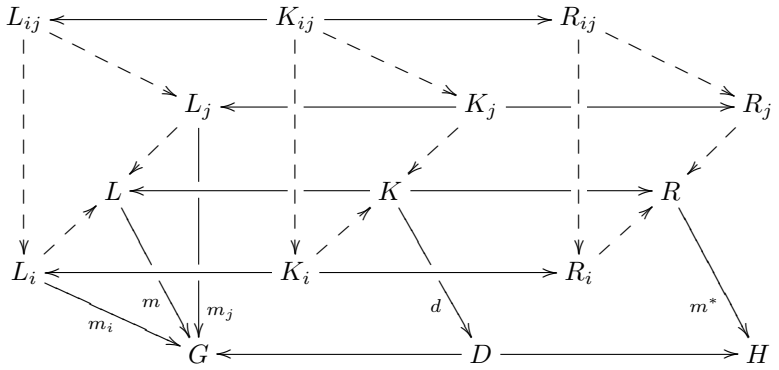


and the corresponding productions $q_{120}, r_{120}, q_{240}, r_{240}$. We can cover the start graph α by the left-hand side of p, i. e., α is a pushout-star of the singleton family consisting of the left-hand side of p. The derived graph G_1 is then given by the right-hand side of p. G_1 can be covered using each of the productions $q_0, \ldots, r_{240}$ exactly once:



The used productions are indicated by their name. The derived graph G_2 is the corresponding pushout-star and equals the one of the node replacement Sierpinski grammar.

A related approach called *amalgamated two-level derivations* is presented in [53]. There, productions have the usual form $p_i : L_i \leftarrow K_i \rightarrow R_i$ of Def. 4.9 on page 48. In order to apply a family $(p_i)_{i \in I}$ of productions in parallel to a graph, we have to specify not only the individual matches $(m_i : L_i \rightarrow G)_{i \in I}$ for the productions, but also the common parts $p_{ij} : L_{ij} \leftarrow K_{ij} \rightarrow R_{ij}$ of each pair of productions p_i, p_j so that we can use these common parts to glue the right-hand sides R_i. The common parts are not fixed in advance, but are chosen on the basis of a given match. The following commutative diagram illustrates the mechanism.



Dashed quadrangles are pushout-stars. Given matches m_i and common parts $p_{ij} : L_{ij} \leftarrow K_{ij} \rightarrow R_{ij}$ (which have to be compatible with the matches, i. e., it must be possible to construct $m : L \rightarrow G$ such that the left face of the diagram commutes), we can construct L, K, R as pushout-stars. The homomorphisms $K \rightarrow L$ and $K \rightarrow R$ are determined by the universal property of the pushout-star of K. The thus obtained production $L \leftarrow K \rightarrow R$ can be applied via the match m by an ordinary direct derivation if the gluing condition is fulfilled, resulting in the derived graph H. The whole construction is called amalgamated two-level derivation: at the first level, the production $L \leftarrow K \rightarrow R$ is composed as an amalgamation of the productions p_i using their intersections p_{ij}; at the second level, this production is applied to the graph by a sequential direct derivation. Note that the intersections p_{ij} have to be chosen on the basis of the matches m_i so that the whole amalgamated production $L \leftarrow K \rightarrow R$ depends on the matches.

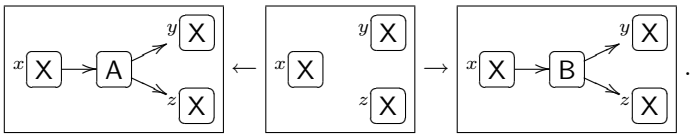
An application of amalgamated two-level derivations is the algebraic specification of a production with edNCE-like embedding. Consider a production $q : \boxed{\mathsf{A}} \leftarrow \emptyset \rightarrow \boxed{\mathsf{B}}$ which shall be applied at a single A node in a graph. Furthermore, the new B node shall receive all edges of the old A node. The production q alone does not specify such a redirection of edges, and due to the gluing condition q is in fact even not applicable if the A node has edges. But we can specify two additional productions q_i, q_o for the edge transfer of incoming and outgoing edges:

$$q_i : \boxed{{}^{n}\boxed{\mu} \rightarrow \boxed{\mathsf{A}}} \leftarrow \boxed{{}^{n}\boxed{\mu}} \rightarrow \boxed{{}^{n}\boxed{\mu} \rightarrow \boxed{\mathsf{B}}},$$

$$q_o : \boxed{\boxed{\mathsf{A}} \rightarrow {}^{n}\boxed{\mu}} \leftarrow \boxed{{}^{n}\boxed{\mu}} \rightarrow \boxed{\boxed{\mathsf{B}} \rightarrow {}^{n}\boxed{\mu}}.$$

μ stands for an arbitrary node label, a precise definition thereof is possible by type graphs with inheritance (Sect. 4.5.2 on page 73). q_i, q_o have a match for each edge incident with the A node, this yields the families $(p_i)_{i \in I}, (m_i)_{i \in I}$ of productions and matches. All matches overlap in the single A node and also the right-hand sides have a common B node, so that the original production q may

be used to described the common parts p_{ij}. As an example, the amalgamated production $L \leftarrow K \rightarrow R$ for an A node with a single incoming and two outgoing edges to X nodes is



A sequential derivation using this amalgamated production correctly implements the embedding of the new B node.

4.3.2 Implicit Connection Mechanisms

A quite general form of implicit connection mechanisms for node replacement grammars is studied in [130, 131]. It is based on *operators* which yield for every node of the host graph a set of related nodes of the host graph (e. g., its neighbours). Now an edge between two nodes of different successor graphs is created if the predecessor nodes are mutually contained in the sets yielded by associated operators. The latter are determined by the used productions, i. e., each production specifies for each node of its successor graph a set of operators which are applied to the single predecessor node. One can imagine these operators as half-edges which have to match each other for an edge to be created in the derived graph. The following definitions are slightly more general than the original ones, first of all the definition of operators is generalized for simplification purposes.

Definition 4.38 (operator). *An* operator A *is given by a family of mappings* $A_G : G_V \rightarrow \mathcal{P}(G_V)$ *for each graph* G *such that* $n \notin A_G(n)$ *for all* $n \in G_V$.

This definition includes operators which yield the direct neighbourhood of a node, but also more general cases such as all nodes which can be reached by a path of specific edge and node labels.

Definition 4.39 (production with operators). *A* production with operators $p : \mu \xrightarrow{\sigma,\tau} R$ *is given by a node label* μ*, a graph* R *and two finite sets* σ, τ *of* connection transformations $c = (A, \gamma, w)$ *where* A *is an operator,* $\gamma \in \Lambda_E$ *an edge label and* $w \in R$ *a node of the right-hand side.*

Definition 4.40 (parallel graph grammar with operators). *A* parallel graph grammar with operators $\mathcal{G} = (\alpha, P)$ *is given by a* start graph α *and a set* P *of productions with operators.*

Definition 4.41 (direct derivation). *Let* $\mathcal{G} = (\alpha, P)$ *be a parallel graph grammar with operators, and let* G, H *be graphs. Then there is a (direct)* derivation *from* G *to* H *within* $\mathcal{G}$ *if all of the following holds:*

1. *There is, for each node $v \in G_V$, a production $p^v : \mu^v \xrightarrow{\sigma^v,\tau^v} R^v \in P$ with $\mu^v = G_\lambda(v)$ such that H_V is isomorphic (as a discrete graph) to $\bigsqcup_{v\in G_V} R_V^v$, the isomorphism being given by $h : \bigsqcup_{v\in G_V} R_V^v \to H_V$.*
2. *Let $\pi : H_V \to G_V$ denote the mapping which assigns to each $n \in H_V$ the predecessor node $v \in G_V$, i. e., $n \in h\big(R_V^{\pi(n)}\big)$. The set of edges of H is given by*

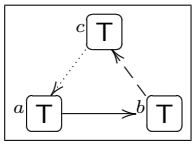
$$\begin{aligned} H_E = &\bigcup_{v\in G_V} \{h(R_E^v)\} \\ &\cup \Big\{(s,\gamma,t) \,\Big|\, \exists (S,\gamma,h^{-1}(s)) \in \sigma^{\pi(s)}, (T,\gamma,h^{-1}(t)) \in \tau^{\pi(t)} : \\ &\qquad \pi(t) \in S_G(\pi(s)) \wedge \pi(s) \in T_G(\pi(t))\Big\} \end{aligned}$$

with the abbreviation for edges $h\big((s,\beta,t)\big) = \big(h(s),\beta,h(t)\big)$.

This definition states that the nodes of G are completely covered by left-hand sides of productions. The derived graph contains the disjoint union of all right-hand sides, plus edges between successor nodes if the predecessor of the source node has a connection transformation in σ which yields the predecessor of the target node and if the predecessor of the target node has a connection transformation in τ which yields the predecessor of the source node.

Example 4.42 (Sierpinski grammar). The node replacement Sierpinski grammar (example 4.32 on page 61) can be reformulated as a parallel graph grammar with operators. Let N_γ^d denote the operator such that $N_{\gamma,G}^d(n)$ yields all nodes of G_V which are connected with n by a γ-typed non-loop edge, n being the source if $d = \text{out}$ and the target otherwise. Then the single production

$$p \;:\; \mathsf{T} \xrightarrow{\sigma,\tau}$$



$$\begin{aligned} \sigma &= \big((N_{\mathsf{e}_0}^{\text{out}}, \mathsf{e}_0, b), (N_{\mathsf{e}_{120}}^{\text{out}}, \mathsf{e}_{120}, c), (N_{\mathsf{e}_{240}}^{\text{out}}, \mathsf{e}_{240}, a)\big)\,, \\ \tau &= \big((N_{\mathsf{e}_0}^{\text{in}}, \mathsf{e}_0, a), (N_{\mathsf{e}_{120}}^{\text{in}}, \mathsf{e}_{120}, b), (N_{\mathsf{e}_{240}}^{\text{in}}, \mathsf{e}_{240}, c)\big) \end{aligned}$$

is equivalent to the node replacement Sierpinski grammar. The connection transformations σ, τ ensure that old edges connecting two triangle nodes are transferred to the correct successor nodes.

A simpler formalism for implicit connection mechanisms is defined and studied in [86, 87], it translates the edNCE mechanism of Sect. 4.2.1 on page 46 to parallel graph grammars Again we have a node replacement grammar, but now each production is equipped with a set of connection instructions. We give a simplified definition which excludes the possibility to change the direction of edges.

Definition 4.43 (edNCEp production). *An* edNCEp production $p : \mu \xrightarrow{e} R$ *is given by a node label* μ*, a graph* R *and a finite set* $e = \{c_1, \ldots, c_n\}$ *of* connection instructions $c_i = (\sigma, \tau, \gamma/\delta, d) \in \Lambda_V \times \Lambda_V \times \Lambda_E \times \Lambda_E \times \{\text{in}, \text{out}\}$.

Definition 4.44 (edNCEp grammar). *An* edNCEp grammar $\mathcal{G} = (\alpha, P)$ *is given by a* start graph α *and a set* P *of edNCEp productions.*

Definition 4.45 (direct derivation). *Let* $\mathcal{G} = (\alpha, P)$ *be an edNCEp grammar, and let* G, H *be graphs. Then there is a (direct)* derivation *from* G *to* H *within* $\mathcal{G}$ *if all of the following holds:*

1. *There is, for each node* $v \in G_V$*, a production* $p^v : \mu^v \xrightarrow{e^v} R^v \in P$ *with* $\mu^v = G_\lambda(v)$ *such that* H_V *is isomorphic (as a discrete graph) to* $\bigsqcup_{v \in G_V} R_V^v$*, the isomorphism being given by* $h : \bigsqcup_{v \in G_V} R_V^v \to H_V$.
2. *Let* $\pi : H_V \to G_V$ *denote the mapping which assigns to each* $n \in H_V$ *the predecessor node* $v \in G_V$*, i. e.,* $n \in h\big(R_V^{\pi(n)}\big)$*. The set of edges of* H *is given by*

$$\begin{aligned} H_E = {} & \{h(R_E^v)\} \\ & \cup \left\{ (s, \delta, t) \,\middle|\, (\pi(s), \gamma, \pi(t)) \in G_E, (H_\lambda(s), H_\lambda(t), \gamma/\delta, \text{out}) \in e^{\pi(s)} \right\} \\ & \cup \left\{ (s, \delta, t) \,\middle|\, (\pi(s), \gamma, \pi(t)) \in G_E, (H_\lambda(t), H_\lambda(s), \gamma/\delta, \text{in}) \in e^{\pi(t)} \right\} . \end{aligned}$$

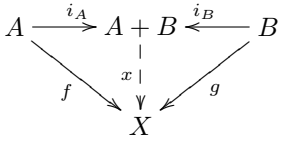
Thus, the connection mechanism connects a pair of nodes s, t of H by an edge labelled δ if the predecessor nodes were connected by an edge labelled γ and if a connection instruction which corresponds to the node and edge labels is defined for s or t. This is different to the operator approach where an edge is created only if both s and t yield matching half-edges.

4.4 Parallelism

Within the algebraic approaches (but also for other formalisms), parallelism of derivations has been studied from another point of view, namely concerning the question whether sequential derivations can be computed in parallel without changing the result [43, 31, 44]. Parallel computations can be described either by an interleaving model where the individual actions are applied sequentially in some order, but the result must not depend on the chosen order, or by truly parallel computations without any order of application and intermediate results.

For the truly parallel model the *parallelism problem* [31] is to find conditions which ensure that a parallel derivation can be sequentialized and vice versa. Within the algebraic approaches, this problem has a nice solution. In order to formulate the solution, we have to define parallel productions and independence properties between derivations. We start with the definition of a categorical coproduct as the basis for the definition of parallel productions.

Definition 4.46 (coproduct). *Let $A, B \in \mathrm{Obj}$ be two objects of a category. A* binary coproduct *is given by a coproduct object $A + B$ and arrows $A \xrightarrow{i_A} A+B \xleftarrow{i_B} B$ which satisfy the universal property of a binary coproduct: for all objects X and arrows $A \xrightarrow{f} X \xleftarrow{g} B$ there exists a unique arrow $A + B \xrightarrow{x} X$ such that the diagram*

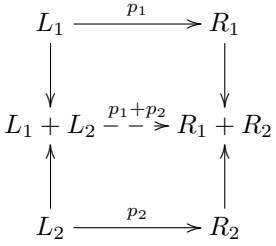


commutes.

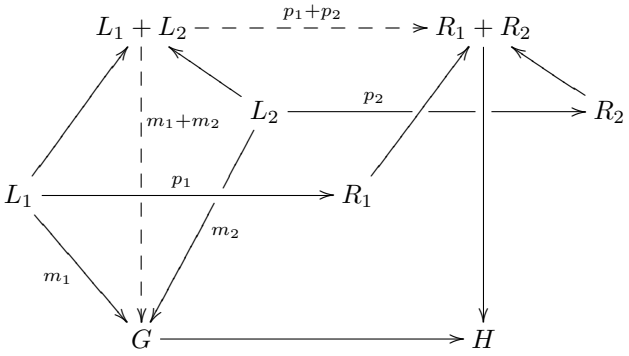
Remark 4.47. By definition, if a coproduct exists, it is unique up to isomorphism. In order to simplify the following considerations, we assume that a suitable coproduct has been fixed uniquely. In **Set**, such a coproduct is the disjoint union with inclusions i_A, i_B. This construction extends to **Graph** and $\mathbf{Graph_P}$. The definition of a binary coproduct can be extended to coproducts of more than two objects.

Now the categorical framework allows a straightforward definition of parallel productions and derivations based on coproducts. This corresponds to the intuitive meaning of a parallel production $p_1 + p_2$ to be the disjoint union of both productions.

Definition 4.48 (parallel SPO production). *Let $p_1 : L_1 \to R_1$ and $p_2 : L_2 \to R_2$ be SPO productions. The* parallel production $p_1 + p_2$ *is defined uniquely by the following commutative diagram in $\mathbf{Graph_P}$, where the vertical parts are coproducts:*



Definition 4.49 (parallel SPO derivation). *Let G a graph, $p_1 : L_1 \to R_1$ and $p_2 : L_2 \to R_2$ be SPO productions with corresponding matches $m_i : L_i \to G$. A* direct parallel derivation *using p_1+p_2 via m_1+m_2, $G \xRightarrow{p_1+p_2, m_1+m_2} H$, is a direct derivation using the parallel production p_1+p_2 via the match m_1+m_2, the latter being uniquely determined by the coproduct $L_1 \leftarrow L_1 + L_2 \to L_2$.*

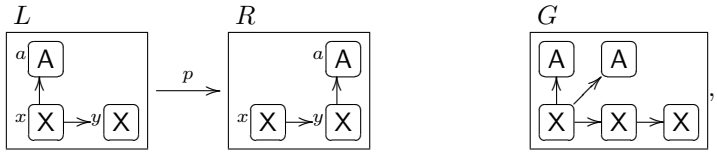


This definition is extended in the obvious way to more than two productions and matches, resulting in the direct parallel derivation using $\sum_i p_i$ *via* $\sum_i m_i$.

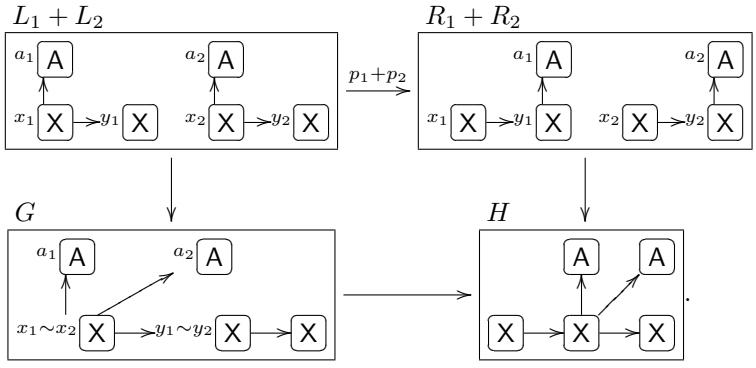
Remark 4.50. Such a parallel derivation can be seen as a special amalgamated two-level derivation (compare with the diagram on page 65, but note that there the DPO approach is used) if we simply set the intersection productions to be $p_{ij} : \emptyset \leftarrow \emptyset \rightarrow \emptyset$. Then pushout-stars like L are just coproducts $\sum_i L_i$.

Remark 4.51. Even if all individual matches m_i are injective, the match $\sum_i m_i$ of the parallel derivation is not injective if the matches overlap.

Example 4.52 (parallel SPO derivation). Given the following production and host graph



there are two matches of L in G. The parallel SPO derivation using p twice via those two matches is



Now we define parallel and sequential independence properties between derivations.

Definition 4.53 (parallel independence of derivations). *A direct SPO derivation* $G \overset{p_2,m_2}{\Longrightarrow} H_2$ *is* weakly parallel independent *of another derivation* $G \overset{p_1,m_1}{\Longrightarrow} H_1$ *if* $m_2(L_2) \cap m_1(L_1) \subseteq m_1(\mathrm{dom}_P\, p_1)$. *Two derivations are* parallel independent *if they are mutually weakly parallel independent.*

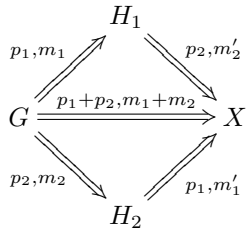
This states that if a derivation is weakly parallel independent of another, their matches do only overlap in objects which are preserved by the latter so that the first derivation can be delayed after the second. If they are parallel independent, each of them can be delayed after the other.

Definition 4.54 (sequential independence of derivations). *Let* $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2'}{\Longrightarrow} X$ *be two consecutive direct SPO derivations.* $H_1 \overset{p_2,m_2'}{\Longrightarrow} X$ *is* weakly sequentially independent *of* $G \overset{p_1,m_1}{\Longrightarrow} H_1$ *if* $m_1^*(R_1) \cap m_2'(L_2) \subseteq m_1^*(p_1(\mathrm{dom}_P\, p_1))$. *If furthermore* $m_1^*(R_1) \cap m_2'(L_2) \subseteq m_2'(\mathrm{dom}_P\, p_2)$, *the two-step derivation is* sequentially independent.

So a second derivation is weakly sequentially independent of a first if it does not depend on objects generated by the first one and, thus, could be applied before the first one.

Now one can prove the following theorem [47]:

Theorem 4.55 (weak parallelism theorem).



1. *Let* $G \overset{p_1+p_2,m_1+m_2}{\Longrightarrow} X$ *be a direct parallel derivation such that* $G \overset{p_2,m_2}{\Longrightarrow} H_2$ *is weakly parallel independent of* $G \overset{p_1,m_1}{\Longrightarrow} H_1$. *Then there exists a match* m_2' *and a direct SPO derivation* $H_1 \overset{p_2,m_2'}{\Longrightarrow} X$ *such that* $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2'}{\Longrightarrow} X$ *is weakly sequentially independent.*
2. *Let* $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2'}{\Longrightarrow} X$ *be a weakly sequentially independent derivation. Then there exists a parallel direct derivation* $G \overset{p_1+p_2,m_1+m_2}{\Longrightarrow} X$ *such that* $G \overset{p_2,m_2}{\Longrightarrow} H_2$ *is weakly parallel independent of* $G \overset{p_1,m_1}{\Longrightarrow} H_1$. □

The weak parallelism theorem relates the interleaving model of parallel computations with the truly parallel model. Within the SPO approach, every

weakly sequentially independent derivation can be transformed in an equivalent parallel derivation, but only parallel derivations where one part is weakly parallel independent of the other may be sequentialized. This is different in the DPO approach where every parallel derivation can be sequentialized into a sequentially independent derivation and vice versa. Thus in the DPO approach the interleaving model and truly parallel model allow to represent the same amount of parallelism, while in the SPO approach the parallel model can express a higher degree of parallelism [31]. This can be seen either as an advantage or as a source of unsafety, the latter because the effect of a parallel derivation cannot be understood by isolated consideration of its components [47].

4.5 Extensions of the Graph Model

Labelled graphs as in Def. 4.1 on page 44 were originally considered as the main notion of graphs. Meanwhile, the necessity of more complex graph models has become apparent. For example, in object-oriented programming a type hierarchy is defined: every object is an instance of its class, but also of all (direct and transitive) supertypes of this class. If we consider nodes as objects, their classes correspond to labels in the traditional graph model. Now the principle of polymorphism states that an instance of a subtype of a type T can appear wherever an instance of T is expected. This is in conflict with the traditional graph model which has no notion of a label hierarchy and within which a match for a node labelled T has to be a node exactly labelled T and not a node of some sublabel of T.

Graph schemas are another important concept which is missing in the traditional graph model. A graph schema specifies which types of edges are allowed from a node of some type to a node of some further type and may additionally contain restrictions such as the multiplicity of edges, i. e., the allowed number of edges of some type at a single node. A popular notation for graph schemas is given by UML class diagrams [136].

Last, but not least, practical applications of graphs and graph rewriting need attributed graphs which associate attribute values with nodes and edges. This reflects the fact that not everything can be represented by structure only, within structural objects non-structural information is stored by means of attribute values.

Solutions to these shortcomings of the traditional graph model are presented within the context of the algebraic approach in [44] based on earlier work. We will discuss these solutions in the following.

4.5.1 Typed Graphs

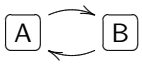
The concept of a graph schema can be implemented by a *type graph*. This is a distinguished (meta-)graph whose nodes play the role of node labels of

the actual graphs and whose edges specify the allowed relationships. A type graph is thus a direct representation of a graph schema. A given actual graph is schema-consistent if there is a homomorphism to the type graph, the graph is then called a *typed graph*. These considerations are formalized by the following definition, which does not yet include inheritance.

Definition 4.56 (type graph, typed graph). *A* type graph *is a graph T whose set of nodes defines the alphabet of node labels, $\Lambda_V = T_V$, and whose labelling function is the identity. A* typed graph $G = (G_V, G_E, G_\tau)$ *over T is a graph (G_V, G_E, G_λ) together with a graph homomorphism $G_\tau : G \to T$. As a consequence, G_λ is uniquely determined by G_τ, namely $G_\lambda = G_\tau|_{G_V}$.*

Definition 4.57 (typed graph homomorphism). *Let G, H be typed graphs over the same type graph, a* typed graph homomorphism *is a graph homomorphism $f : G \to H$ such that $H_\tau \circ f = G_\tau$.*

In the context of typed graphs, one rather speaks of node and edge types than of node and edge labels. As an example for a type graph, let us construct the type graph of all bipartite graphs with a single edge label $*$. A bipartite graph is a graph whose nodes can be divided into two disjoint sets such that every edge starts in a node of one of the sets and ends in a node of the other set. The type graph T^{bi} is given by $T_V^{\text{bi}} = \{\mathsf{A}, \mathsf{B}\}$, $T_E^{\text{bi}} = \{(\mathsf{A}, *, \mathsf{B}), (\mathsf{B}, *, \mathsf{A})\}$:



Now a graph is bipartite if and only if there exists a homomorphism to T^{bi}.

The type graph of an ordinary, labelled graph (Def. 4.1 on page 44) has the complete set $\Lambda_V \times \Lambda_E \times \Lambda_V$ as the edges, i. e., for every edge label and every pair of nodes s, t (including $s = t$) there is a corresponding edge. This reflects the fact that Def. 4.1 provides no means to restrict the usage of edges to edge-specific node types.

On the basis of a typed graph, productions and derivations in the double-pushout and single-pushout approach can be defined by prefixing graphs and homomorphisms with the attribute 'typed' [44]. We do not give these definitions here, since they are a special case of the definitions of section Sect. 4.5.3 on page 75.

4.5.2 Typed Graphs with Inheritance

The object-oriented concept of inheritance between types is not reflected in the definition of a type graph. In this section, we define a representation of inheritance for typed graphs. In [44], inheritance is restricted to node types, and productions are defined in terms of a mapping from single abstract productions to sets of concrete productions without inheritance. This hides inheritance from the category and its graph homomorphisms at the cost of a

technical construction. Conversely, [139] integrates the inheritance relation for node and edge types within the category, but there is no notion of a type graph. In the following, we use the ideas of both approaches to define inheritance for node and edge types within a category of typed graphs. A similar construction is presented in [56].

Definition 4.58 (inheritance relation). *Let T be a type graph. An* inheritance relation $\leq$ *on T is a pair $(\leq_V, \leq_{\Lambda_E})$ of partial orders on T_V and Λ_E, respectively. a is called a* subtype *of b if $a \leq_V b$ or $a \leq_{\Lambda_E} b$ (so a type is a subtype of itself). $\leq_V$ and $\leq_{\Lambda_E}$ define a partial order on the set $T_V \times \Lambda_E \times T_V$: $(s', \beta', t') \leq_E (s, \beta, t) \Leftrightarrow s' \leq_V s \wedge \beta' \leq_{\Lambda_E} \beta \wedge t' \leq_V t$.*

Definition 4.59 (type graph with inheritance). *A* type graph with inheritance *is a tuple $(T, \leq, \mathcal{A})$ where T is a type graph, $\leq$ an inheritance relation on T, and $\mathcal{A} \subseteq T_V \cup \Lambda_E$ a set of* abstract types *of T.*

The simplest way to integrate such a type graph with inheritance into the previous framework is to flatten it into a type graph without inheritance, i. e., to copy all β-typed edges connecting nodes s, t to β'-typed edges connecting nodes s', t' where β', s', t' are subtypes of β, s, t, respectively [44].

Definition 4.60 (closure of type graph with inheritance). *Let $(T, \leq, \mathcal{A})$ be a type graph with inheritance. Its* closure *is given by the type graph $\overline{T}$ having*

$$\overline{T}_V = T_V \ ,$$
$$\overline{T}_E = \{(s, \beta, t) \in T_V \times \Lambda_E \times T_V \,|\exists e \in T_E, (s, \beta, t) \leq_E e\} \ .$$

The concrete closure *$\widehat{T}$ is the largest subgraph of $\overline{T}$ without abstract types, i. e., $\widehat{T}_V = T_V \setminus \mathcal{A}$, $\widehat{T}_E \subseteq T_E \setminus (T_V \times \mathcal{A} \times T_V)$.*

Definition 4.61 (typed graph with inheritance). *Let $(T, \leq, \mathcal{A})$ be a type graph with inheritance. A typed graph over $\overline{T}_E$ ($\widehat{T}_E$) is a* (concrete) typed graph with inheritance *over $(T, \leq, \mathcal{A})$.*

The explicit definition of concrete typed graphs is necessary for the definition of productions: left-hand sides of productions may contain nodes of abstract types because they just serve as a pattern, but host graphs and results of a derivation have to consist of nodes of concrete types exclusively.

Now inheritance has to be integrated in the definition of a homomorphism. Considering the example of a match $m : L \to G$, it follows from the principle of polymorphism that candidates for a match of an object x of the left-hand side have to be of a subtype of the type of x, $G_\tau(m(x)) \leq L_\tau(x)$. This is exactly the condition for an inheritance-aware homomorphism:

Definition 4.62 (typed graph homomorphism with inheritance). *Let G, H be typed graphs with inheritance, a* typed graph homomorphism with inheritance *$f : G \to H$ is a pair of functions $(f_V : G_V \to H_V, f_E : G_E \to H_E)$ such that $f_E((s, \beta, t)) = (f_V(s), \gamma, f_V(t))$ with some $\gamma \in \Lambda_E$ and $(H_\tau \circ f)(x) \leq G_\tau(x)$ for all $x \in G$ (where this means, as usual, the node or edge component depending on whether x is a node or an edge).*

4.5.3 Typed Attributed Graphs with Inheritance

The key idea to represent attributed graphs is to reuse the graph model for this purpose: attribute values are indicated by edges from the attributed objects to value nodes. For example, a real-valued attribute 'length' of a node having value 3.14 is represented by an edge of type 'length' from the node to the value node 3.14. In order to formally define an attributed graph, some preliminary definitions concerning signatures and algebras are needed [44]:

Definition 4.63 (signature). *A* signature $\Sigma = (S, OP)$ *consists of a set* S *of* sorts *and a family* $OP = (OP_{w,s})_{(w,s)\in S^*\times S}$ *of* operation symbols*. For* $op \in OP_{w,s}$*, we write* $op : w \to s$ *or* $op : s_1 \dots s_n \to s$*. If* $|w| = 0$*,* $op :\to s$ *is a* constant symbol*.*

A signature is a syntactical description of an algebra. For example, within the natural numbers $\mathbb{N}_0$ the operations $+$ and $\cdot$ are defined, and the numbers 0 and 1 play a special role. This can be abstracted to the signature NAT which has a single sort nat and the operation symbols $+ : nat \times nat \to nat$, $\cdot : nat \times nat \to nat$, $\mathbb{O} :\to nat$, $\mathbb{I} :\to nat$. The natural numbers are then an implementation of this signature, a NAT-algebra:

Definition 4.64 (Σ-algebra). *Let* $\Sigma = (S, OP)$ *by a signature. A* Σ-algebra $A = ((A_s)_{s\in S}, (op_A)_{op\in OP})$ *is defined by a* carrier set A_s *for each sort* s *and a* mapping $op_A : A_{s_1} \times \dots \times A_{s_n} \to A_s$ *for each operation symbol* $op : s_1 \dots s_n \to s$*. For constant symbols,* $op_A :\to A_s$ *is a constant* $c_A \in A_s$*.*

For $\mathbb{N}_0$ as an implementation of NAT, this means $A_{nat} = \mathbb{N}_0$, $+ : (x, y) \mapsto x + y$, $\cdot : (x, y) \mapsto xy$, $\mathbb{O}_A = 0$, $\mathbb{I}_A = 1$.

Definition 4.65 (Σ-algebra homomorphism). *Let* A, B *be* Σ*-algebras. A* homomorphism $f : A \to B$ *is a family* $(f_s)_{s\in S}$ *of mappings* $f_s : A_s \to B_s$ *such that* $f_s(op_A(x_1, \dots, x_n)) = op_B(f_{s_1}(x_1), \dots, f_{s_n}(x_n))$ *for all operation symbols* $op : s_1 \dots s_n \to s$ *and all* $x_i \in A_{s_i}$*.*

Definition 4.66 (final Σ-algebra). *Let* Σ *be a signature. Its* final Σ-algebra Z *is given by* $Z_s = \{s\}$ *for each sort and* $op_Z : (s_1, \dots, s_n) \mapsto s$ *for each operation symbol* $op : s_1 \dots s_n \to s$*.*

For every Σ-algebra A, there is a unique homomorphism $z_A : A \to Z$, it simply maps a value of a carrier set A_s onto its sort s.

From now on, we fix a specific data signature Σ and denote its final algebra by Z_Σ. The following definition of an attributed graph differs from [44] in that we do not allow attributed edges. The latter require attribute edges starting at edges which makes an extension of the usual graph model necessary. Our simplification is justified by the fact that we do not make use of attributed edges in subsequent chapters.

Definition 4.67 (attributed graph). *An* attributed graph *G is given by $G = (G_N, G_E, G_\lambda, G_D)$ where $G_D = ((G_{D,s})_{s\in S}, (op_{G_{D,s}})_{op\in OP})$ is a Σ-algebra, (G_V, G_E, G_λ) with $G_V = G_N \cup \bigcup_{s\in S} G_{D,s}$ is a graph, and $G_E \subseteq G_N \times \Lambda_E \times G_V$, i. e., source nodes of edges are in G_N.*

This definition states that every possible element of the carrier sets $G_{D,s}$ is a node in G, and that these nodes may only be used as targets of edges. Such an edge is to be interpreted as an attribute edge, the value being the target node which is drawn from one of the carrier sets (e. g., the natural or real numbers). However, this definition does not make explicit which attribute is encoded by an attribute edge.

Definition 4.68 (attributed graph homomorphism). *Let G, H be attributed graphs, an* attributed graph homomorphism *$f : G \to H$ is a graph homomorphism such that the family $(f|_{G_{D,s}})_{s\in S}$ is an algebra homomorphism.*

Similar to the definition of a typed graph with inheritance in the previous section, we define a typed attributed graph with inheritance by requiring an attributed graph homomorphism to a distinguished type graph.

Definition 4.69 (attributed type graph with inheritance). *A type graph with inheritance $(T, \leq, \mathcal{A})$ is an* attributed type graph with inheritance *if T is an attributed graph and $\leq_V$ is the equality on carrier sets, i. e., $a \leq_V b \wedge (a \notin T_N \vee b \notin T_N) \Rightarrow a = b$.*

Definition 4.70 (typed attributed graph with inheritance). *Let $(T, \leq, \mathcal{A})$ be a distinguished attributed type graph with inheritance with algebra Z_Σ. A* typed attributed graph with inheritance *$G = (G_N, G_E, G_D, G_\tau)$ over $(T, \leq, \mathcal{A})$ is both an attributed graph $(G_N, G_E, G_\lambda, G_D)$ and a typed graph with inheritance (G_V, G_E, G_τ) over $(T, \leq, \mathcal{A})$. $G_\lambda = G_\tau|_{G_V}$ is uniquely determined by G_τ.*

These definitions exclude inheritance on carrier sets and, thus, on types of attributes. Of course, in object-oriented programming attributes of objects may be objects themselves. The apparent conflict can be resolved by treating object values of an attribute as ordinary nodes. Types of true attributes in the sense of Def. 4.67 are restricted to non-object-types. This different treatment can be justified by the following consideration: elements of the carrier sets are required to exist exactly once in an attributed graph, they cannot be created or deleted. In the case of real numbers, there has to be a single node for every real number. This reflects the fact that a given real number exists only once – if two real numbers are equal, they are the same. On the other hand, two objects of the same type and with identical attribute values are equal, but not necessarily the same – objects have an identity. This identity is expressed by being an ordinary node which may be created or deleted.

An attributed type graph with inheritance $(T, \leq, \mathcal{A})$ is a graph schema which contains for every attribute a of a type C an attribute edge labelled

a from the node C to the corresponding sort of the attribute value. A typed attributed graph with inheritance G over $(T, \leq, \mathcal{A})$ must not have attributes which are not defined by $(T, \leq, \mathcal{A})$. However, the definition does not specify that for each object of G all attributes allowed by the type graph are present, nor does it guarantee that the same attribute is present at most once per object. *Graph constraints* can be added which ensure that each object has all allowed attributes exactly once [44]. According to [56], we give a definition which characterizes graphs with such desirable properties:

Definition 4.71 (strict and complete typed attributed graphs). *A typed attributed graph with inheritance G over $(T, \leq, \mathcal{A})$ is* strict *if for each node $n \in G_N$ and each edge $(G_\tau(n), \beta, t) \in \overline{T}_E$ with $t \in S$ there exists at most one edge $(n, \beta, t') \in G_E$ such that $G_\tau((n, \beta, t')) = (G_\tau(n), \beta, t)$. G is* complete *if there exists exactly one such edge for each node n.*

In order to instantiate the SPO approach for typed attributed graphs with inheritance, we have to define homomorphisms and the corresponding category, and we have to ensure the existence of pushouts. We start with total homomorphisms.

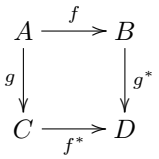
Definition 4.72 (typed attributed graph homomorphism with inheritance). *Let G, H be typed attributed graphs with inheritance, a* typed attributed graph homomorphism *is an attributed graph homomorphism $f : G \to H$ such that $H_\tau \circ f \leq G_\tau$.*

Proposition 4.73 (category AGraph$(T, \leq, \mathcal{A})$). *Let $(T, \leq, \mathcal{A})$ be an attributed type graph with inheritance. Typed attributed graphs with inheritance over $(T, \leq, \mathcal{A})$ and their total homomorphisms define a category* **AGraph**$(T, \leq, \mathcal{A})$. □

Definition 4.74 (class $\mathcal{M}$ of type-preserving monomorphisms). *A homomorphism $f : G \to H$ is called* type-preserving *if $H_\tau \circ f = G_\tau$. Given an attributed type graph with inheritance, the class of type-preserving monomorphisms is denoted by $\mathcal{M}$.*

Proposition 4.75 (pushouts in AGraph$(T, \leq, \mathcal{A})$). **AGraph**$(T, \leq, \mathcal{A})$ *has pushouts along $\mathcal{M}$-homomorphisms (i. e., if at least one of the given homomorphisms is in $\mathcal{M}$), and $\mathcal{M}$-homomorphisms are closed under pushouts (i. e., for a given homomorphism $f \in \mathcal{M}$, the resulting homomorphism f^* of the pushout is also in $\mathcal{M}$).*

Proof. The construction of the pushout follows the construction in Prop. 4.13 on page 50. Let f, g of the following square be given with $f \in \mathcal{M}$.



The node component D_V of D is constructed in **Set**, i. e., $D_V = (B_V \sqcup C_V)/\sim^*$ where $\sim^*$ is the least equivalence relation which identifies $f(a)$ and $g(a)$ for each $a \in A_V$. Here, the equivalence class of $c \in C$ is given by $[c] = f(g^{-1}(c)) \sqcup \{c\} \subseteq B_V \sqcup C_V$. In order to prove this we have to show the minimality of the set $[c]$ because $\sim^*$ is defined as the least equivalence relation induced by the relation $f(a) \sim g(a)$. If we extend the set by $c' \in C$ with $c' \neq c$ and $c' \sim [c]$, there must be some $b' \in f(g^{-1}(c))$ with $c' \sim b'$ and, thus, some a' with $c' = g(a')$ and $b' = f(a')$. It follows that $f(a') = b' \in f(g^{-1}(c))$ and, because f is a monomorphism, $a' \in g^{-1}(c)$. But then $c' = g(a') = c$, which is a contradiction. Similarly, if we extend the set by $b' \in B$ with $b' \sim [c]$, it follows $b' = f(a') \sim g(a') \in \{c\} \Rightarrow b' \in f(g^{-1}(c)) \subset [c]$, i. e., the set $[c]$ is minimal. As a consequence, the construction of D_V amounts to $D_V = (B_V \setminus f(A_V)) \sqcup C_V$.

Now we set $f^*(c) = c$ and $g^*(b) = g(f^{-1}(b))$ if $b \in f(A_V)$, otherwise $g^*(b) = b$. For $d \in B_V \setminus f(A_V)$, we set $D_\tau(d) = B_\tau(d)$. For $d \in C_V$, we set $D_\tau(d) = C_\tau(d)$. This already ensures the node component part of the assertion $f^* \in \mathcal{M}$, and the requirement $(D_\tau \circ g^*)(b) \leq B_\tau(b)$ is fulfilled for all nodes $b \in B_V$ since for $b \notin f(A_V)$ we have $(D_\tau \circ g^*)(b) = D_\tau(b) = B_\tau(b)$ and for $b = f(a)$ we have $(D_\tau \circ g^*)(b) = D_\tau(g(a)) = C_\tau(g(a)) \leq A_\tau(a) = B_\tau(f(a)) = B_\tau(b)$.

It remains to construct the edge component part. For $e = (s, \beta, t) \in C_E$, we set $f^*(e) = (f^*(s), \beta, f^*(t))$. For $e = (s, \beta, t) \in B_E$, we set $g^*(e) = f^*(g(f^{-1}(e)))$ if $e \in f(A_E)$, otherwise we set $g^*(e) = (g^*(s), \beta, g^*(t))$. Finally D_E is chosen as required by f^*, g^*, i. e., $D_E = f^*(C_E) \cup g^*(B_E)$. Note that it may happen that an edge of $f(A_E)$ is mapped by g^* to the same edge in D_E as an edge of $B_E \setminus f(A_E)$.

The resulting mappings f^*, g^* have the properties of typed attributed graph homomorphisms with inheritance, furthermore we have $f^* \in \mathcal{M}$, and the square is commutative. The pushout property can be verified by the following observations: the construction of the node components of D, f^*, g^* is the same as in **Set**, the construction of the edge components is the minimal one which leads to a commutative square, the construction of the type homomorphism D_τ ensures that the type is always the greatest of all possible types. □

Now the category $\mathbf{AGraph}(T, \leq, \mathcal{A})$ and its properties can be used to define the category $\mathbf{AGraph_P}(T, \leq, \mathcal{A})$ which has partial homomorphisms as arrows.

Proposition 4.76 (category $\mathbf{AGraph_P}(T, \leq, \mathcal{A})$). *Let $(T, \leq, \mathcal{A})$ be an attributed type graph with inheritance. Typed attributed graphs with inheritance over $(T, \leq, \mathcal{A})$ and their partial homomorphisms define a category $\mathbf{AGraph_P}(T, \leq, \mathcal{A})$.* □

Proposition 4.77 (pushouts in $\mathbf{AGraph_P}(T, \leq, \mathcal{A})$). *$\mathbf{AGraph_P}(T, \leq, \mathcal{A})$ has pushouts along $\mathcal{M}_P$-homomorphisms (partial homomorphisms which are $\mathcal{M}$-homomorphisms on their domain), and $\mathcal{M}_P$-homomorphisms are closed under pushouts.*

Proof. The construction of the pushout is the construction of Prop. 4.22 on page 53, translated to $\mathbf{AGraph_P}(T, \leq, \mathcal{A})$. We use Prop. 4.75 on page 77 for the three pushouts of the construction, noting that for each pushout we have at least one $\mathcal{M}$-homomorphism given. The final coequalizer construction of Prop. 4.21 on page 53 directly carries over. □

In order to really make use of attributes within graph rewriting, i. e., within a derivation, computations on attribute values of nodes of a match must be possible, resulting in attribute values of nodes of the right-hand side. Within a production, such a computation is encoded as a formula which contains attribute values of the left-hand side as variables. This is reflected by the following definitions [44].

Definition 4.78 (variables and terms). *Let $\Sigma = (S, OP)$ be a signature and $X = (X_s)_{s \in S}$ a family of sets, the* variables *of sort s. The family $T_\Sigma(X) = (T_{\Sigma,s}(X))_{s \in S}$ of* terms *is inductively defined:*

- $X_s \subset T_{\Sigma,s}(X)$,
- $op(t_1, \ldots, t_n) \in T_{\Sigma,s}(X)$ *for each operation symbol (including constant symbols) $op : s_1 \ldots s_n \to s$ and all terms $t_i \in T_{\Sigma,s_i}(X)$.*

Definition 4.79 (term algebra). *The family of terms $T_\Sigma(X)$ defines the* term algebra *over Σ and X, where the carrier set of a sort s is given by $T_{\Sigma,s}(X)$ and the mapping of an operation symbol $op : s_1 \ldots s_n \to s$ by $op_{T_{\Sigma,s}(X)} : (t_1, \ldots, t_n) \mapsto op(t_1, \ldots, t_n)$.*

Definition 4.80 (typed attributed SPO production with inheritance). *A* typed attributed SPO production with inheritance *$p : L \to R$ is a partial typed attributed graph monomorphism with inheritance, where L, R are typed attributed graphs with inheritance which share the same term algebra $T_\Sigma(X)$ for some family X of variables, and where $p|_{T_{\Sigma,s}(X)}$ is the identity for all sorts s.*

The last requirement ensures that the application of a production does not modify the value nodes. Note that in [44] only the double-pushout approach is presented, but since we will need single-pushout productions in the sequel, we give the corresponding SPO definitions.

Definition 4.81 (typed attributed match with inheritance). *A* match *for a production $p : L \to R$ in a typed attributed host graph with inheritance G is a total typed attributed graph homomorphism with inheritance $m : L \to G$.*

Definition 4.82 (direct typed attributed SPO derivation with inheritance). *Let $p : L \to R$ be a production and $m : L \to G$ a match in a typed attributed host graph with inheritance G. A* direct derivation *using p via m, denoted as $G \overset{p,m}{\Longrightarrow} H$, is given by the following single-pushout diagram in the category $\mathbf{AGraph_P}(T, \leq, \mathcal{A})$:*

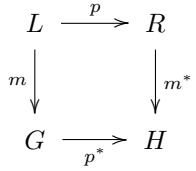


Figure 4.4 shows an example for a derivation which squares a real-valued attribute. The variable v is required to match the current value of the attribute, i. e., the node v of L matches the node 2 of the host graph G. The redirection of the attribute edge from v to $v \cdot v$ removes the edge to 2 and creates an edge to 4 in the host graph via the pushout construction.

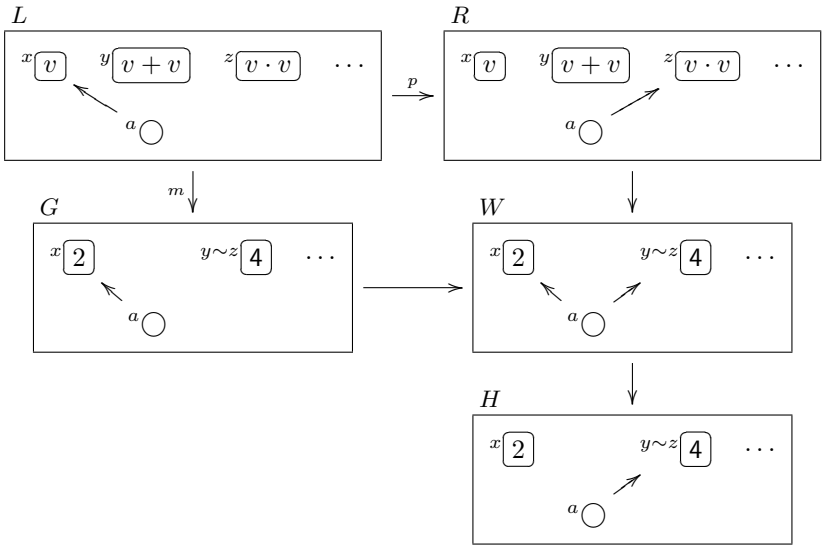


Figure 4.4. Typed attributed SPO derivation: the attribute indicated by the attribute edge is squared. In fact, every term for L, R and every real number for G, W, H are part of the graphs, and m maps each term to its value. For obvious reasons, the complete graphs cannot be depicted, omitted parts are indicated by ...

4.6 High-Level Replacement Systems

The definitions of the algebraic approaches to graph rewriting (pushout, pullback, relation-algebraic) can be stated without reference to a concrete graph model. In fact, they can be given for any category which satisfies certain conditions such as the existence of pushouts for derivations. This leads to the notion

of *high-level replacement systems* whose definitions are as abstract as possible. For the double-pushout approach, the whole theory can be generalized from graphs to weak adhesive HLR categories [44]. These are characterized by the following properties:

1. There is a class $\mathcal{M}$ of monomorphisms which are closed under isomorphisms, composition and decomposition.
2. Pushouts and pullbacks along $\mathcal{M}$-homomorphisms exist, and $\mathcal{M}$-homomorphisms are closed under pushouts and pullbacks.
3. Pushouts along $\mathcal{M}$-homomorphisms are weak van Kampen squares (see [44]).

4.7 Programmed Graph Replacement Systems

Programmed graph replacement systems are an extension of basic graph grammars by means to control the application of productions. There is no longer a monolithic set of productions to which the current graph is subjected at every derivation step, the set of productions to apply is rather selected by some sort of control flow. This is similar to the idea of table L-systems (Sect. 3.5 on page 24). For an overview of programmed graph replacement systems, see [173].

Several techniques exist to prescribe the control flow. There are simple ones which are suitable for theoretical studies, but may turn out to be too restrictive for practical use. As an example, *programmed graph grammars* use a control flow diagram whose nodes specify sets of productions which have to be applied when the control flow reaches a node [21]. Each node has two sets of outgoing edges which prescribe the path taken by the control flow: depending on whether a production could be applied or not, an edge of the first or the second set is used.

On the other side of the spectrum of programmed graph replacement systems, the control flow is governed by imperative control structures. This gives maximal freedom to the user, but may not be as qualified for theoretical investigations. A tight coupling between control structures and productions is generally desirable. For example, there should be branching or iterating control structures whose conditions are linked with the success or failure of the (previous or potential) application of productions [116, 201] or even complex transactions, which are composed of productions and control structures. For sequential grammars with their nondeterministic match selection among all possible matches, the control structures should preserve the nondeterminism as far as possible, which is in conflict with deterministically working traditional control structures. For example, we could think of a control structure which executes either of two transactions with nondeterministic choice, or of a control structure which executes two transactions in arbitrary order [173]. This has to be implemented in such a way that, if a nondeterministic choice

leads to a dead-end, another choice is tried. The common technique is to use backtracking for this purpose which in this case has to be extended from the single choice of a match for a production to a whole sequence of statements consisting of control structures, productions and transactions.

4.8 Graph Rewriting Software

This section presents four different graph rewriting tools. Of course, there are a lot of further graph rewriting tools. An extensive but a bit out-of-date list is contained in [133], a recent comparison of tools on the basis of three case studies is given by [164] (see also Sect. 10.8 on page 347).

4.8.1 PROGRES

The PROGRES language and programming environment is probably the most elaborated tool for graph transformations which has been implemented up to now [173, 175, 188, 151]. It uses directed attributed graphs as data model. The semantics is that of logic-based structure replacement (Sect. 4.2.7 on page 58), embedded in imperative control structures (Sect. 4.7 on the previous page). Within PROGRES, a graph schema with inheritance and edge cardinalities is specified which is used for type-checking of productions at compile-time and, together with integrity constraints, for the current graph at run-time. Derived attributes can be specified and are computed using an incremental attribute evaluation algorithm.

Productions are specified by their left- and right-hand sides as usual, together with textual specification of attribute transfer and edNCE-like embedding (Sect. 4.2.1 on page 46). The matching of left-hand sides allows advanced features like optional nodes which need not exist, node sets which are mapped onto a non-empty set of nodes, and negative nodes or edges which must not exist. The PROGRES language also defines queries which have the same syntax as left-hand sides of productions and can be used for subgraph tests.

Productions are the basic building blocks and can be combined by control structures to define transactions. These contain nondeterministic ones as discussed in Sect. 4.7 on the previous page. A whole transaction has the same characteristics as a single production, including nondeterministic choices and the initiation of backtracking if a choice leads into a dead-end.

A complete program is a mixture of visual and textual elements, where visual elements are used to represent queries and the left- and right-hand sides of productions. The programming environment of PROGRES consists, among others, of a mixed textual and visual editor, a compiler which creates code for an abstract graph transformation machine, an implementation of such a machine to execute a specification within the PROGRES programming environment, and back ends to translate the abstract machine code to C or Modula-2 in order to obtain a prototype from the PROGRES specification.

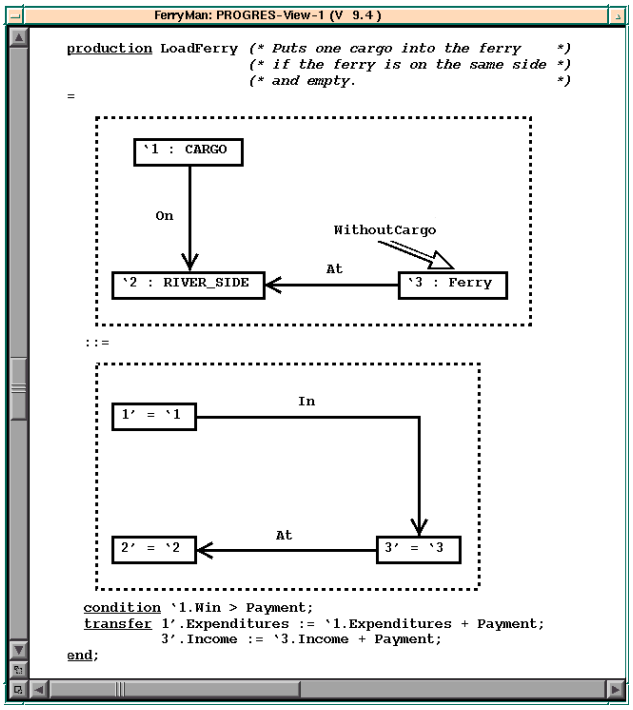


Figure 4.5. Screenshot of PROGRES environment (from [151])

Fig. 4.5 shows a screenshot displaying a production which loads a ferry. The structural part is specified visually, while the quantitative part (conditions on and assignments of attribute values) is specified textually.

4.8.2 AGG

The Attributed Graph Grammar (AGG) system [55, 44, 192] is an integrated development tool for typed attributed graph transformation implemented in Java. Graphs are similar to those of Def. 4.70 on page 76, but inheritance is defined differently and parallel edges of the same type are allowed. The consistency of graphs with respect to their type graph is checked, also graph constraints which have to be fulfilled can be specified. Nodes and edges can be attributed with Java objects. Differently from the theory, each object has exactly one value for each of its attributes – the theory does not enforce this, but it can be expressed by corresponding graph constraints. Also the usage of Java objects in itself differs from the theory since their classes are parts of a inheritance hierarchy, but the definition based on Σ-algebras does not handle inheritance for carrier sets of attribute values (see the discussion on page 76 in Sect. 4.5.3).

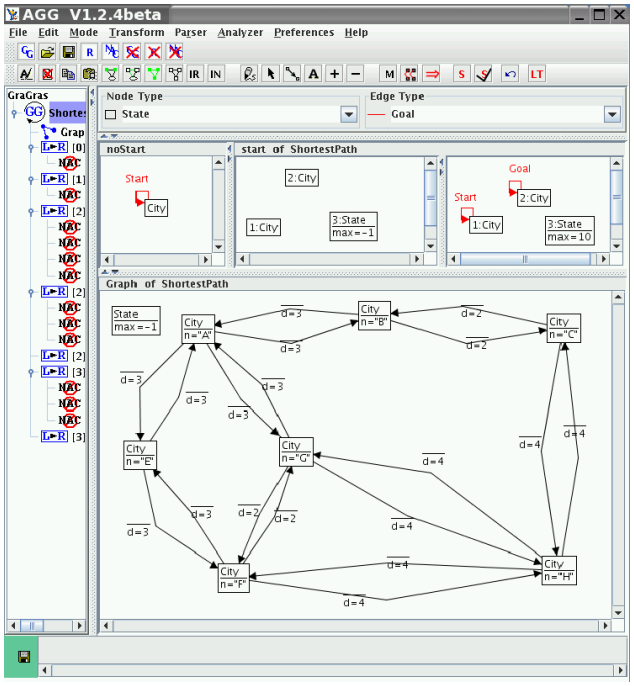


Figure 4.6. Screenshot of the AGG software (from [192])

Productions are treated according to the single-pushout approach. Application conditions can be specified including boolean Java expressions. The double-pushout approach can be obtained as a built-in feature by activation of the gluing condition as application condition.

AGG works sequentially. A single (possibly noninjective) match is selected for actual application out of all possible matches, either randomly or by preference, e. g., by user interaction. On application, the graph structure is modified according to the theory. The expressions for the computation of new attribute values may contain invocations of arbitrary Java methods, this obviously goes beyond the theory since it may trigger arbitrary side effects.

A simple control flow is provided by the concept of layers. Each layer has a priority and contains a set of productions. Starting with the layer of lowest priority, productions of the current layer are applied as long as possible. Afterwards, the next layer with respect to priority is made current. If all layers have been processed, the transformation terminates.

AGG provides several analysis techniques. The conformance to graph constraints can be checked. Critical pairs can be detected (minimal examples of parallel dependent derivations), this is important in order to verify that a graph grammar is confluent, i. e., that the final result after productions have been applied as long as possible does not depend on the order of production

applications. A graph parser checks whether a given input graph is part of a graph language specified by a set of parsing rules and a stop graph. It tries to find a derivation using the parsing rules from the input to the stop graph. Furthermore, for a subset of graph grammars AGG can decide if they are terminating or not.

As an integrated development tool, AGG has a graphical user interface with visual editors for types, graphs and productions. The core functionality of graph rewriting is provided by a graph transformation engine which is independent of the visual environment. It may also be used by other tools.

4.8.3 GrGen.NET

GrGen.NET is a relatively new graph transformation tool for the .NET framework [65]. A type graph with inheritance can be specified by defining node and edge types as well as connection assertions, and the compliance of a graph with the type graph can be checked on demand. Attribute and type conditions can be specified for patterns. The sequential derivation mechanism is based on the single-pushout approach. Several productions may be composed with logical and iterative control of application. The invocation is done either interactively within a shell-like environment, or in custom programs by including the GrGen.NET library.

The authors of GrGen.NET attach great importance to a fast implementation of the pattern matching algorithm which finds occurrences of left-hand sides in the current graph. For a pattern composed of several subpatterns, the order in which subpatterns are matched has the largest impact on the time efficiency of the whole matching algorithm. An estimate of the matching time can be described heuristically by a cost model, and then an optimization algorithm can be used to find an optimal order with respect to the cost model. This order defines a *search plan* ([7] and Sect. 7.3.1 on page 197) which governs the later execution of the matching algorithm. By an implementation of such a technique in GrGen.NET, the tool was able to outperform any other tested graph transformation tool by at least one order of magnitude [64]. An important feature of the underlying cost model of GrGen.NET is its dependence on statistics of the current graph like the number of nodes and edges of specific types. This leads to a relatively precise cost model, so that the computed search plan is really a good choice.

4.8.4 vv

The vv software [177, 178] differs from most other graph rewriting tools in that its target is the modification of polygon meshes, which can be seen as special graphs. For the vv software, the representation of polygon meshes by *graph rotation systems* was chosen among the multitude of possible polygon mesh representations (see [59] for examples). A graph rotation system consists of a set V of vertices and, for each vertex $v \in V$, a circular list v^* of its

neighbours such that the symmetry condition $w \in v^* \Leftrightarrow v \in w^*$ is fulfilled. A graph rotation system induces an undirected graph if we let $v, w \in V$ be connected by an edge if and only if $w \in v^*$. Furthermore and more important, it induces a unique closed topological polygon mesh by considering all circular sequences $v_0, \ldots, v_n = v_0$ as polygons for which all v_{i+1} immediately follow v_{i-1} in the circular list v_i^*. If in addition we have positional information about the vertices, we obtain a geometric polygon mesh.

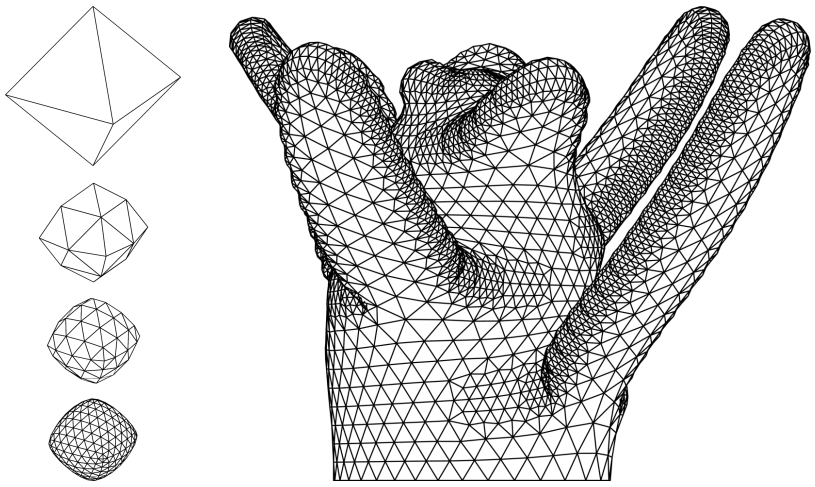
Now the vv software provides a set of useful operations (called the *vertex-vertex algebra*) on graph rotation systems like the insertion of a vertex in the neighbourhood of another vertex, its removal, or the determination of the vertex which follows a given vertex in the neighbourhood of another given vertex. These operations are implemented as part of a vv library, and they are also made available directly by the syntax of the vv language. The vv language is an extension of C++ and is implemented by a special source code preprocessor which transforms its input into C++ source code. The language follows the imperative programming paradigm; thus, it is *not* rule-based. The following source code is the implementation of the Loop subdivision scheme which is an algorithm for surface subdivision (from [178]), the 3D outcome is shown Fig. 4.7(a) on the next page.

```
// splits existing edge between p and q by insertion of new vertex
vertex insert(vertex p, vertex q) {
    vertex x;
    make {p, q} nb_of x;  // sets circular list of neighbours of x
    replace p with x in q;// replaces p with x in list of q's neighbours
    replace q with x in p;
    return x;
}

// implements Loop subdivision
void loop(mesh& S) {
    synchronize S; // create a local copy of S, accessible via prefix ‘
    mesh NV;
    forall p in S {
        double n = valence p; // number of neighbours of p
        double w = (5.0/8.0)/n - pow(3.0/8.0 + 1.0/4.0*cos(2*PI/n),2)/n;
        p$pos *= 1.0 - n*w;
        forall q in ‘p {
            p$pos += w * ‘q$pos
            if (p < q) continue; // vertices are ordered
            vertex x = insert(p, q);
            // nextto a in b yields vertex next to a in list of b's,
            // neighbours, prevto correspondingly
            x$pos = 3.0/8.0 * (‘p$pos + ‘q$pos)
                    + 1.0/8.0 * ‘(nextto q in ‘p)$pos
                    + 1.0/8.0 * ‘(prevto q in ‘p)$pos;
            add x to NV;
        }
```

```
    }
    forall x in NV {
        vertex p = any in x; // choose any neighbour of x
        vertex q = nextto p in x;
        make {nextto x in q, q, prevto x in q,
              nextto x in p, p, prevto x in p} nb_of x;
    }
    merge S with NV; // add vertices of NV to S
}
```

A special feature of vv is the use of the **synchronize** statement together with backquote-prefixed vertex expressions. The **synchronize** statement creates a local read-only copy of the current mesh. If `v` is a vertex of the current structure which already existed when the copy was made, `‘v` denotes its corresponding copy. This provides a means to consistently refer to the old structure while modifying the current structure step by step. Note that this problem could be solved in a more elegant way by a suitable parallel grammar operating on graph rotation systems.

Figure 4.7(b) shows the relevance of the vv approach for plant modelling. It displays the outcome of a model for an apical meristem [177]. The meristem grows by cell subdivision at the tip. Regulated by concentrations of an inhibitor on an active ring just below the tip, cells differentiate and become the germs of primordia, which grow out laterally. The model simplifies the real situation by considering only the surface of the apical meristem and primordia, so that cells are two-dimensional.



(a) (b)

Figure 4.7. Polygon meshes modelled with vv (from [177]): (**a**) initial mesh and three iterations of Loop subdivision; (**b**) apex with primordia

5

Relational Growth Grammars

While the previous chapters report the current situation in L-system and graph transformation theory and application, from this chapter on we present the extensions which have been developed as part of this thesis. They are all linked with the formalism of *relational growth grammars* which is a generalization of parallel graph grammars and incorporates L-systems as a special case.

5.1 Introduction

The aim of this thesis, design and implementation of a graph grammar based language for functional-structural plant modelling (FSPM), already anticipates that graph grammars are a suitable formalism for FSPM. Nevertheless, we have to justify this. Since graph grammars can be seen as an extension of L-systems, they inherit the proved strengths of the latter. But as we will see in the next paragraphs, L-systems have several drawbacks which can be overcome easily with the help of graph grammars.

At first sight, the ongoing success of L-systems in plant modelling is evidence for their fitness for this purpose. This holds especially for pure structural modelling as can be seen by the fact that there was basically no attempt to extend the notion of context-sensitive, bracketed L-systems with respect to structure since its appearance. The string encoding is easily understandable and well-suited for plant-like structures as long as there is at most a simple kind of interaction within the structure, and the rule-based programming paradigm often provides a direct translation of botanical rules.

On the other hand, a variety of extensions of L-systems with respect to function have been defined. With 'function' we mean nonstructural parts in order to reflect the complementarity of function and structure which is connoted by the term FSPM. Parametric L-systems can be seen as the first functional extension of the basic L-system formalism because they allow the representation of internal data within the structure. Differential, environmentally-

sensitive and open L-systems as well as sensitive growth grammars and arithmetical-structural operators can also be classified as functional extensions since they focus on parameters and three-dimensional interpretation of the structure and not on its pure topology. L-systems with programming statements use parameter values as basis for their computations, and approaches like L+C generalize the type of parameters from numeric types to arbitrary C++ types.

Despite their success, L-systems have shown some deficiencies in their nearly 40 years of use in plant modelling [100]:

- 3D structures have to be serialized into strings of symbols in order to make them representable by an L-system. This is acceptable for tree-like structures, but it hinders the modelling of more complex topologies.
- Strings produced by an L-system have to be interpreted geometrically by the turtle. This implies a sort of semantic gap and an extra step of processing between the rewriting formalism and 3D structures, in contrast to modern data structures for 3D world (e. g., scene graphs [59, 82, 84]) where information is represented more directly. The semantic gap is especially noticeable if there is a complex interaction based on the 3D geometry, e. g., collision or competition for light.
- In a structure generated by an L-system, basically only two relations between symbols can be modelled: 'successor' between consecutive symbols and 'branch' where a branch in brackets comes out. In many applications, particularly for multiscaled models, it is desirable to have more relations at hand.
- L-systems give little support for computations on the created structure (e. g., determination of the mass of a plant or a branch): the pure string representation does not provide a means for efficient structure-aware navigation due to the indirect encoding of branches by bracket symbols. In fact, arithmetical-structural operators of sensitive growth grammars are not defined on the string representation, but on the result of turtle interpretation which is a true tree.

Based on these considerations, it seems natural for an extension of L-systems to retain the basic rule-based principle, but to use trees or even graphs as representation: then we can dispense with brackets as part of the internal string representation and directly represent branchings in the structure. This allows a straight navigation which is the basis for efficient implementations of sensitivity such as arithmetical-structural operators. If we extend the structure anyway, we can at the same time enrich it internally from parameterized symbols to nodes which are instances of classes in the object-oriented sense. Such nodes may provide additional information besides their parameters, e. g., the location in three-dimensional space, which then can be accessed easily within expressions. Turtle commands have a natural translation to this setting: they become nodes of a 3D scene graph. Turtle commands which only move the turtle become transformation nodes, turtle commands which create geometry

become shape nodes. This closes the semantic gap between the structure and its interpretation: nodes may be both elements of the structure and 3D objects by their very nature.

The usage of true graphs and graph grammars instead of just trees and tree grammars further increases the expressiveness of the formalism and opens the door for applications in a wide range of fields:

- Metabolic and gene regulatory networks appear at a low level of plant development. The program transsys (Sect. 3.15.4 on page 38) combines L-systems with such networks. However, both parts are specified separately there, each with an own sublanguage. If networks can be represented directly as a graph, their dynamics can be implemented by a graph grammar together with structural parts at higher levels of the plant. Neural networks are a further type of networks in biological or biology-inspired modelling.
- Cellular automata (Sect. 2.4 on page 14) are defined on n-dimensional grids. Such a grid can be implemented as a graph where edges specify direct neighbours, i. e., the topology of the grid. Thus, a grid and its dynamics have a direct representation as graphs and graph grammars, whereas this would become intricate within the formalism of L-systems.
- The grid of a cellular automaton is a prototype of the representation of a discretized world. The example of artificial ants in Sect. 2.5 on page 15 makes use of such a world, but it could also be applied in the context of plant modelling to represent the soil on which several plants grow and compete for resources. This can even be generalized to three-dimensional grids, or to non-regular structures like polygonal meshes which could represent, e. g., cell layers and tissues.
- Finally, arbitrary relationships between entities of the model can be established by edges. These can also be scale crossing, leading to multiscale models [67].

Given this list of applications, it is in fact somewhat astonishing that true graph grammars have hardly been used in plant modelling, with the exception of early studies in the 1970s in the wake of the evolution of L-systems [118]. As the biological knowledge increases, the integration of the mentioned issues in plant models becomes more and more important. We thus believe that it is about time to dispense with the traditional L-system representation of a string of symbols, and to use graphs and graph grammars instead. Even Lindenmayer himself states in the introduction of [118]:

> The one-dimensional character of basic L system models and the rigid grid structure of cellular automata constructs certainly limit the modelling possibilities of both of these theories. The very recently evolved theory of parallel graph generating and related systems tries to remove both obstacles and therefore constitutes a promising approach.

The transition to graphs comes at nearly no cost, since it is backward compatible and the increase in required computer memory for the representation

of graphs instead of strings poses no problem for modern computers, possibly with the exception of model simulations where an extremely large number of nodes is created.

But the transition to graphs and graph grammars alone would only be half the way to a suitable extension of the L-system formalism which is able to build the basis of FSP models with increasing complexity within the next years: this transition is a structural extension and has to be completed by an adequate functional extension. This has to be consistently designed in such a way that it covers the whole heterogeneous range of existing functional L-system extensions, and it should at the same time be simple and comprehensive as a programming language formalism. Here, we can learn from the evolution of imperative programming languages [13]: initially, such languages were very technical low-level languages without any built-in support for the management of, e. g., control flow or larger systems. The introduction of structured, modular and object-oriented techniques led to simple but comprehensive high-level languages which assist the programmer to keep complex software intelligible, extensible and maintainable.

The basic L-system formalism is a low-level formalism whose semantics is the plain application of productions to a string, comparable to the semantics of low-level languages whose programs consist of sequences of instructions (with **goto** and **call** being the sole control flow instructions). In [13], it is expected that the evolution of L-systems will give birth to object-oriented and even visual programming techniques for L-systems, and an extension of L-systems is already presented which makes use of the object-oriented principles of inheritance and polymorphism for symbols. The combination with a conventional imperative and object-oriented programming language like C++ in the case of L+C is a different branch of the evolution of L-systems: the typing system of symbols is not object-oriented, but C++ is used to define types of parameters, computations within productions and whole functions.

If we take a further step along the evolution of L-systems by unifying both branches on the basis of graphs and additionally allowing the imperative part to control the application of productions like in programmed graph replacement systems (Sect. 4.7 on page 81), we can achieve the following properties:

- The control flow is controlled like in any structured imperative programming language, i. e., there are statements like **if** or **for**.
- The control flow invokes the application of productions. There is no monolithic set of productions which are always applied jointly; sets of productions are rather statements in the sense of imperative programming languages which are executed when they are reached by the control flow. This generalizes table L-systems (Sect. 3.5 on page 24) and many approaches of programmed graph replacement systems in a convenient way (at least from a practical point of view, the underlying theory loses elegance due to the unrestricted inclusion of imperative programming).

- Right hand-sides of productions may contain imperative programming statements which are executed as a side-effect of production application. They can be used, e. g., for auxiliary calculations in order to compute new attribute values for nodes. Together with the previous item, this encompasses the possibilities of L-systems with programming statements (Sect. 3.10 on page 29).
- The global program structure is that of an object-oriented program. Thus, we have a class hierarchy with inheritance and polymorphism. A subset of the classes can be used as node types, this subhierarchy is reflected in the type graph with inheritance (Sect. 4.5.2 on page 73). Statements (and, thus, sets of productions) constitute the bodies of methods.

In doing so, we follow the proposal of [107] where the possibilities and advantages of a combination of the imperative, object-oriented and rule-based paradigms were discussed in the context of ecological and plant modelling.

As a completing step, we have to provide a means to conveniently access the structure not only as part of the actual graph rewriting process, but also as part of general expressions by a *query language*. This can be seen as a reflection of the structure within the functional part since it enables us to easily specify computations within the structure. As we noted in the list of deficiencies of L-systems, one might be interested in the total mass of all descendant objects of a given node, or in the node of some specific type which is closest in three-dimensional space to a given node. Rather than having to hand-code these functions in the imperative part of the formalism by loops, recursive methods and other technical necessities, a query language allows to concisely specify what to search for (e. g., all descendant objects), and *aggregate functions* can be used to compute aggregated values on the basis of query results (e. g., the sum of masses of a set of objects). A concrete query language could reuse the syntax of the left-hand side of productions since both search for occurrences of a pattern in the current structure. Query expressions in combination with aggregate functions generalize arithmetical-structural operators of growth grammars, see Sect. 3.11 on page 30. Furthermore, we can also specify globally sensitive functions like those of growth grammars (Sect. 3.11 on page 30) within our model in a versatile way. For example, while the globally sensitive function which computes the total biomass of all elementary units within a vertical cone starting at a given position had to be implemented as a fixed part of the GROGRA software, we can specify this as an aggregate function applied to a result of a query, given that the modelling environment provides us with the quite basic geometric function which tests whether an object lies within a cone. By this means, query expressions and aggregate functions allow us to combine universal "atomic" building blocks provided by the modelling environment in order to specify arbitrarily complex sensitive functions within the model itself. This technique was already proposed in [107], it shifts the borderline between the model itself and the modelling environment or some external software in such a way that the model-specific parts

can easily and favourably be implemented as part of the model. Interestingly, the evolution of imperative programming languages has also integrated query expressions recently in the language C# [128] (a stable version is expected in the course of the year 2008); however, our approach is independent thereof.

We believe that graph grammars, interwoven with an imperative language and the possibility of structural queries within expressions, all based on object-oriented principles, are a good candidate for the definition of a state-of-the-art FSPM language. Such a language extends the proved techniques of L-systems for the structural part and of imperative programming for the functional part in a way which takes into account the fact that a whole is more than the mere sum of its parts:

- The structural view of L-systems is retained but extended to graphs in order to be able to directly represent arbitrary relationships, e. g., complex functional relationships like interaction networks.
- The procedural view of functional models is reflected in the imperative language. The latter is augmented by query expressions as a technique for computation and information flow; this technique assists in expanding functional models from a single entity to the whole plant structure created by the structural part.

In this chapter, we define the formalism of *relational growth grammars* (RGG for short) as a formal basis for such a language. We concentrate on the definition of the rewriting mechanism and specify the imperative part in a rather abstract way by a general control flow (and by Σ-algebras for the predefined functions of the modelling environment). Part II presents the XL programming language, a concrete programming language for which the RGG formalism can be implemented easily; there it is also shown how graph grammars and imperative programming can be interwoven in practice. The name 'relational growth grammars' has been chosen in view of the intended principal application, in order to stress the continuousness of the development from L-systems via growth grammars to the RGG formalism, and to emphasize its being based on relational notions like graphs and relations (see also Sect. 5.7 on page 114).

The presented RGG formalism and its usage for the modelling of (artificial) living systems are in the spirit of the following opinion of Kari, Rozenberg and Salomaa [90], but are at the same time an addendum:

> However, one feature very characteristic for the architecture of all living beings is that life is fundamentally parallel. A living system may consist of millions of parts, all having their own characteristic behavior. However, although a living system is highly distributed, it is massively parallel.
> Thus, any model for artificial life must be capable of simulating parallelism – no other approach is likely to prove viable. Among all grammatical models, L systems are by their very essence the most suitable

> for modeling parallelism. L systems may turn out to be even more suitable for modeling artificial life than real life.

The addendum is that parallel graph grammars are even more suitable for modelling parallelism of (artificial) living systems than L-systems. This will be shown in Chap. 10.

5.2 Graph Model

The first question to be answered for the definition of relational growth grammars is that of the graph model. A seamless integration of an imperative object-oriented programming language requires a typing system for graphs which reflects the principles of inheritance and polymorphism. In order to generalize parameterized symbols and to represent properties of objects, attributed graphs are the natural choice. But what is the precise meaning of nodes and edges, with which features are they equipped? We start our considerations with the case of graph equivalents of L-system strings.

5.2.1 Axial Trees

The data structure of L-systems, a string of symbols, is interpreted by the turtle as an *axial tree* [161, 67] with the help of bracket symbols. Or the other way around, the string is considered as a linearization of an axial tree in postfix notation. An axial tree is a *rooted tree* (a single node is distinguished as root) with two types of directed edges, *branching* edges and *continuing* edges. Each node has at most one incoming edge, an arbitrary number of outgoing branching edges, and at most one outgoing continuing edge. Two symbols which are direct neighbours in the L-system string, possibly separated by complete bracketed sequences, are connected by a continuing edge, the first symbol after an opening bracket is connected with the last symbol before the bracket by a branching edge. A maximal sequence of nodes which are connected by continuing edges is called an *axis*. The botanical interpretation is as follows [67]. The growth of a plant is governed by *apical meristems* which generate, through repeated activity, a sequence of *metamers* in the *apical growth process*. Each metamer consists of an internode, a leaf together with its insertion node, and a lateral meristem. This sequence corresponds to an axis and would be represented in an L-system string by a sequence of symbols on the same level, i. e., without brackets. Depending on the resolution, the symbols stand for whole metamers or the individual organs internode, leaf, node. The created lateral meristems may become apical meristems themselves as part of the *branching process*. The outcome of this process, an axis of higher order, is inserted within brackets in the L-system string.

The graph model corresponding to axial trees uses nodes as the actual objects and edges as the representation of relations between objects. The edge

alphabet contains two elements 'branching' and 'continuing', the correspondingly typed edges will be called branch edge and successor edge and denoted by + and >, respectively, from now on following the notation of [67]. Nodes have attributes, edges do not. Such a graph model is sufficient for the representation of L-system structures. The precise representation is given by the following definitions. We omit the representation of parameters as node attributes for the sake of simplicity. This comes without loss of generality since attributes are internal to nodes, but the following is concerned with structural issues only.

Definition 5.1 (well-nested word). *Let V be an alphabet without % and bracket symbols and set $V_B = V \cup \{[,], \%\}$. The set of all* well-nested words over *V is defined as the language over V_B generated by the context-free (sequential) grammar*

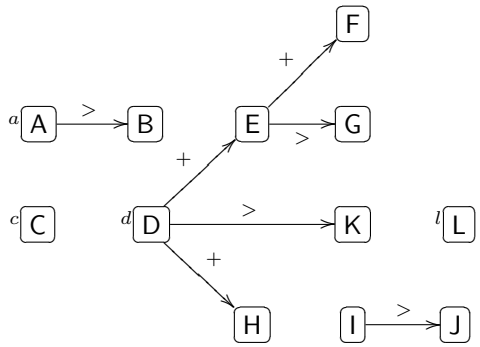
$$\begin{aligned} \alpha &\to A_0 \ , \\ A_0 &\to \varepsilon \ , \\ A_0 &\to A \ , \\ A &\to aA_0 \quad \forall a \in V \ , \\ A &\to \%A_0 \ , \\ A &\to [A_0]A_0 \ . \end{aligned}$$

In this definition, α is the axiom, ε denotes the empty word, the nonterminal A stands for a non-empty well-nested word, while A_0 stands for a possibly empty well-nested word. The symbols [,] and % denote branches and a cutting as explained in Sect. 3.2 on page 19. The fourth production actually stands for a set of productions, namely a single production for each symbol $a \in V$. The definition extends that of [158] by the symbol %. Note that the grammar is $LL(1)$. Thus it is unambiguous and the parse tree for a given well-nested word is unique so that we can recursively define mappings from well-nested words in the manner of synthesized attributes [2]. We immediately make use of this:

Definition 5.2 (graph representation of well-nested words). *The* graph representation *of well-nested words ν over V is given by a mapping $T = (T^G, T^L, T^R, T^B) : \nu \mapsto (G, L, R, B)$ where G is a graph over the alphabet $\Lambda = (V, \{>, +\})$, $L \subseteq G_V \cup \{\%\}$ denotes the* left-most nodes *of G, $R \subseteq G_V \cup \{\%\}$ the* right-most nodes *of G and $B \subseteq G_V$ the* pending branch nodes *of G. T is defined recursively according to the productions of the grammar of Def. 5.1 (a stands for a symbol of V; χ, ψ are well-nested words):*

$$
\begin{aligned}
T(\varepsilon) &= (\emptyset,\emptyset,\emptyset,\emptyset)\ , \\
T(a\chi) &= \Bigg(\{n\} \cup T^G(\chi) \cup \bigcup_{m\in T^L(\chi)\setminus\{\%\}} (n,>,m) \quad \cup \bigcup_{m\in T^B(\chi)} (n,+,m),\ \{n\}, \\
&\qquad \begin{cases} T^R(\chi) : T^R(\chi) \neq \emptyset \\ \{n\} : T^R(\chi) = \emptyset \end{cases},\ \emptyset \Bigg) \ \textit{with some new a-labelled node } n, \\
T(\%\chi) &= \big(T^G(\chi), \{\%\}, \{\%\} \cup T^R(\chi), \emptyset\big)\ , \\
T([\psi]\chi) &= \big(T^G(\chi) \cup T^G(\psi), T^L(\chi), T^R(\chi), T^B(\chi) \cup T^L(\psi) \cup T^B(\psi)\big)\ .
\end{aligned}
$$

It follows that the sets L, R of left- and rightmost nodes contain at most a single node, but may contain an additional % in which case the original word contained a %-symbol at the highest (non-nested) level. L is empty if and only if R is empty. As an example for the representation, the well-nested word [[AB]C]D[[E[F]G]][H%IJ]K[ ]%L is translated to the (unconnected) graph



together with the set of left-most nodes $\{d\}$, the set of right-most nodes $\{\%, l\}$ and the set of pending branch nodes $\{a, c\}$. Note how symbols to the right of % are unconnected with the rest, and that the set of right-most nodes contains % to indicate a %-symbol at highest level. Note also that information is lost: the %-symbol detaches the following nodes from the graph (for L-systems, we could have a rule which replaces % by some non-cutting word), and empty or redundant pairs of brackets and the order of branches cannot be recovered from the graph representation. Under the assumption that typical reasonable L-systems do not replace the symbols [,], % by other symbols and that the order of branches is irrelevant, this loss of information poses no problems. This assumption is reasonable from a botanical point of view.

5.2.2 RGG Graph Model

The model of an axial tree can be embedded in a lot of graph models, be it a simple one like plain graphs according to definition Def. 4.1 on page 44, a more complex one like hypergraphs of Sect. 4.2.2 on page 47 or even hierarchical graphs where entire subgraphs are nested within edges. When choosing

a graph model, a typical trade-off between different requirements has to be made: on one hand, it should be simple enough so that it is intelligible, that its representation within computer memory is efficient, and that its graph instances can be manipulated efficiently. On the other hand, it should allow the direct representation of a sufficiently large class of objects and their relationships.

Because it is not unusual for L-system based models to produce structures of several 100,000 or even millions of objects, special attention has to be paid to the memory-related aspect of the trade-off. It is a reasonable assumption that typical RGG structures consist for a large part of tree-like substructures which require $n-1$ edges for n nodes. If – which is also a reasonable assumption – the types of actually used edges in such trees are drawn from a small set (e. g., mostly branch and successor edges), it would be a striking waste of memory if edges are represented as complex objects with a rich set of features which are hardly used.

Based on these considerations, the graph model of relational growth grammars is chosen to be that of typed attributed graphs with inheritance (Def. 4.70 on page 76) so that attributes and object-oriented principles are reflected in the model. The semantics of edges is to stand for plain relationships between their incident nodes. Thus, they have no attributes, and the restriction that no parallel edges of the same type may exist is reasonable. Concrete edge types are drawn from a finite set $\widehat{\Lambda}_E$ without inheritance relation between them, the types branch and successor are included in $\widehat{\Lambda}_E$. The set Λ_E of all edge types is given by the power set of $\widehat{\Lambda}_E$ with the exclusion of the empty set so that an inheritance relation $\leq_{\Lambda_E}$ can be defined by the subset relation (we identify $T \in \widehat{\Lambda}_E$ with the singleton set $\{T\} \in \Lambda_E$) [139]. This allows specifications like 'an edge of type T or U exists from a to b' for the left-hand side of productions, since 'T or U' corresponds to the abstract edge type $\{T, U\}$ which is a supertype of both T and U. If attributed edges or parallel edges of the same type are really necessary, a common trick helps: an auxiliary node with a single incoming edge from a and a single outgoing edge to b plays the role of such an edge from a to b. We formalize these verbal considerations in the definition of an RGG type graph:

Definition 5.3 (RGG type graph). *Let $\widehat{\Lambda}_E$ be a finite set of* concrete edge types*. An attributed type graph with inheritance $(T, \leq, \mathcal{A})$ over the edge alphabet $\Lambda_E = \mathcal{P}(\widehat{\Lambda}_E) \setminus \{\emptyset\}$ is an* RGG type graph *if $\leq_{\Lambda_E} = \subseteq$ and $\Lambda_E \cap \mathcal{A} = \Lambda_E \setminus \{\{\gamma\} | \gamma \in \widehat{\Lambda}_E\}$, i. e., the abstract edge types are exactly the non-singleton sets.*

Definition 5.4 (RGG graph). *An* RGG graph *is a typed attributed graph with inheritance over an RGG type graph.*

The following definition just serves to simplify subsequent statements.

Definition 5.5 (category RGGGraph). *Let* $(T, \leq, \mathcal{A})$ *be a fixed, distinguished RGG type graph. The category* $\mathbf{AGraph_P}(T, \leq, \mathcal{A})$ *is called* **RGG-Graph** *(over* $(T, \leq, \mathcal{A})$*).*

With the graph model of RGG graphs, sets, lists, axial trees and grids like those of cellular automata are directly representable [98, 99]. These data structures were considered in [107] as important building blocks for the specification of a wide spectrum of ecological phenomena. Metabolic and gene regulatory networks can also be represented, but require auxiliary nodes for edges with numeric parameters which encode, e. g., the strength of inhibition of an enzyme by a substance. Relationships between different scales of a model [67] may be expressed by scale edges. However, in this case hierarchical graphs might be a better choice, but then the method of graph transformation has to be extended correspondingly [22].

5.3 Connection Mechanism

The choice of the basic connection mechanism of relational growth grammars has a great influence on the power, intelligibility and usability of the formalism. The presented mechanisms of Sect. 4.3 on page 59 have to be investigated, but also the technique of parallel SPO derivations (Def. 4.49 on page 69) turns out to be very convenient.

5.3.1 L-System-Style Connection

One requirement is the possibility to specify L-system productions in the RGG formalism. Since left- and right-hand side of an L-system production do not share common objects, embeddings of the gluing type are not directly usable, whereas edNCE-like connection mechanisms seem to be suitable. Namely, if we translate the application of an L-system production $\mathsf{A} \to [\mathsf{B}]\mathsf{CD}$ to the string UAX (with result $\mathsf{U}[\mathsf{B}]\mathsf{CDX}$) to the language of graphs, it implies, for every incoming edge e of the A-node, the creation of embedding edges from the source of e (U-node in the example) to both the C-node and the B-node, the latter edge being of type 'branch', and for every outgoing edge f of the A-node the creation of an embedding edge from the D-node to the target of f (X-node in the example). This creation of embedding edges is edNCE-like as it can be specified by a list of connection instructions of the form $(v, \mu, \gamma/\delta, w, d)$.

However, there is a crucial difference to the edNCE mechanism due to the parallelism of L-systems. The neighbours of a removed node are themselves replaced as part of a direct derivation. A natural idea is to use parallel edNCEp grammars of Def. 4.44 on page 68. But these establish connections based on node types exclusively, while for L-system simulation, we have to distinguish special nodes of right-hand sides as left-most (C-node in the example), right-most (D-node in the example) and pending branch (B-node in the example)

according to the namings of Def. 5.2 on page 96. Now for two neighbouring nodes which are replaced by L-system productions, connection edges have to be established from the right-most node of the successor of the source to the left-most and pending branch nodes of the successor of the target. So both successors are involved in the decision whether a connection edge is created or not. This can be seen as an application of "matching half-edges", namely as a special case of the operator approach to parallel graph grammars (Def. 4.41 on page 66) in the following way:

Definition 5.6 (translation of axis-propagating L-system production). *Let V be an alphabet without % and bracket symbols. An L-system production $p : a \to \chi$ with a well-nested word χ over V and $L \neq \emptyset$ for the graph representation $(G, L, R, B) = T(\chi)$ of χ (Def. 5.2 on page 96) is called* axis-propagating. *Its* translation to a production with operators *is given by $T(p) : a \xrightarrow{\sigma,\tau} G$ with*

$$\sigma = \bigcup_{\gamma \in \widehat{\Lambda}_E, s \in R \setminus \{\%\}} \left\{ (N_\gamma^{\text{out}}, \gamma, s), (N_\gamma^{\text{out}}, +, s) \right\} ,$$

$$\tau = \bigcup_{\gamma \in \widehat{\Lambda}_E, t \in L \setminus \{\%\}} \left\{ (N_\gamma^{\text{in}}, \gamma, t) \right\} \cup \bigcup_{\gamma \in \widehat{\Lambda}_E, t \in B} \left\{ (N_\gamma^{\text{in}}, +, t) \right\} .$$

N_γ^d denotes the operator such that $N_{\gamma,G}^d(n)$ yields all nodes of G_V which are connected with n by a γ-typed non-loop edge, n being the source if $d = \text{out}$ and the target otherwise. In the terms of [130], this operator is of depth 1, and the connection transformations are orientation preserving.

Remark 5.7. The term 'axis-propagating' is in analogy to the usual terminology which calls L-systems propagating if all productions have non-empty right-hand sides. For an axis-propagating production, the right-hand sides have to be non-empty even if we remove all bracketed branches. This means that the current axis 'grows' (except for the case of % as right-hand side which is also axis-propagating, but cuts off the axis).

Theorem 5.8 (equivalence of axis-propagating L-system and translated graph grammar). *Let V, V_B be as in Def. 5.1 on page 96. Let $\mathcal{G} = (V_B, \alpha, P)$ be a D0L-system such that $P = \{\% \to \%, [\to [,] \to]\} \cup P'$ where P' contains only axis-propagating productions. Then $\mathcal{G}$ is equivalent to the translated parallel graph grammar with operators $T(\mathcal{G}) = \left(T^G(\alpha), \{T(p) \,|\, p \in P'\}\right)$ in the sense that for every derivation the diagram*

$$\begin{array}{ccc} \mu & \xRightarrow{\quad\mathcal{G}\quad} & \nu \\ {\scriptstyle T^G}\downarrow & & \downarrow{\scriptstyle T^G} \\ T^G(\mu) & \xRightarrow{\quad T(\mathcal{G})\quad} & T^G(\nu) \end{array}$$

commutes.

Proof. See Appendix 5.A on page 118. □

Remark 5.9. Of course, a true equivalence cannot be obtained because of the loss of information of the graph representation of well-nested words. But if we are only interested in the axial trees which are represented by L-system words, the equivalence holds.

Remark 5.10. Although the theorem is stated for D0L-systems, it should be clear that the extension to nondeterministic or parametric L-systems is straightforward and only requires a more complex notation.

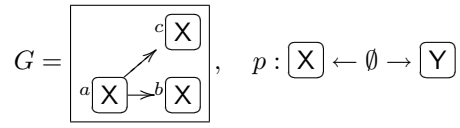
The problem with non-axis-propagating productions like $\mathsf{A} \to [\mathsf{B}]$ is that they may lead to connection edges in the graph representation whose predecessor nodes have an arbitrarily large distance. For example, if we apply the above production together with $\mathsf{X} \to \mathsf{Y}$ to XAAAX, the result is $\mathsf{Y}[\mathsf{B}][\mathsf{B}][\mathsf{B}]\mathsf{Y}$ so that a connection edge between both Y-typed nodes has to be created although their X-typed predecessors have a distance of four. A simple solution would be to add some unused auxiliary symbol to the right-hand side of non-axis-propagating productions and to have a postprocessing step which removes all chains of auxiliary nodes. A more direct solution is to extend the operators N^d_γ so that they skip nodes whose successors do not propagate the axis:

- $N^d_{\gamma,G}(n)$ for $\gamma \neq +$ traverses G starting at n in direction d using only edges of type γ. If it finds a node m whose production p^m within the current derivation is axis-propagating, it stops traversing at m and yields m.
- $N^d_{+,G}(n)$ traverses G starting at n in direction d using edges of an arbitrary type. If it finds a node m whose production p^m within the current derivation is axis-propagating, it stops traversing at m and, if it has traversed at least one single branch edge, yields m.

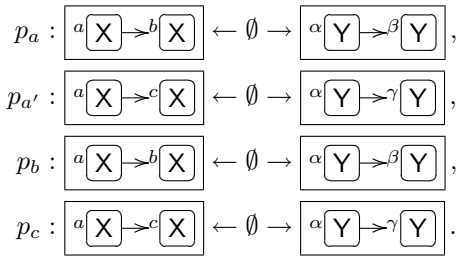
This requires an extension of the definition of parallel graph grammars with operators because operators have to know about the productions which are used for nodes. The proof of equivalence would be even more technical than for Theorem 5.8, but it should be possible.

The question arises whether the connection mechanism which is required for L-systems can also be obtained by algebraic pushout approaches. Unfortunately, these approaches do not provide a straightforward solution. we would have to specify a production which implements the application of the connection mechanism to a match. Its left-hand side would have to include all neighbours of the match which are involved in the connections to be created because algebraic productions require all modified or used objects to be on their left-hand side. But this is not possible because the number of neighbours is not known in advance and one cannot construct a DPO or SPO production with a variable number of nodes on the left-hand side. A solution to this problem within the DPO approach and sequential derivations is presented in [53], it makes use of amalgamated two-level derivations (see page 64). The key idea

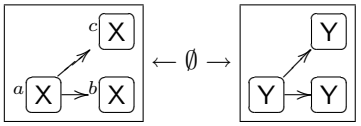
is to extend the original production for every connection instruction and every matching neighbour individually. The resulting single amalgamated production has then the same effect as an edNCE production. However, this technique was only investigated for derivations which are sequential at a macroscopic level, i. e., as a translation of edNCE embedding. Nevertheless, it can also be used for parallel derivations if we extend the original productions not only based on the neighbours within the current graph, but also on the desired neighbours in the derived graph. For example, consider the following graph G and DPO production p with the three obvious matches.



In order to implement the connection mechanism, we have to specify a single production-match pair for each combination of an original match and an embedding edge. The productions have to include both the embedding edges of the host graph on their left-hand side and the new embedding edges of the derived graph on their right-hand side. This yields four productions (the indices indicate from which original match the productions are derived).



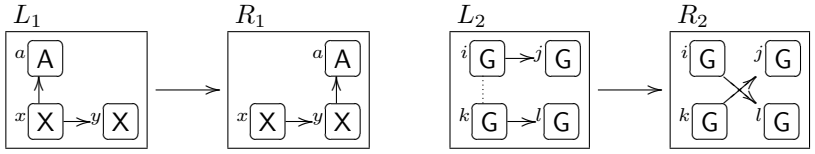
The matches and the intersection productions $p_{ij} : L_{ij} \leftarrow \emptyset \rightarrow R_{ij}$ for amalgamation are indicated by the identifiers $a, \ldots, \gamma$. The resulting amalgamated production (see diagram on page 65) and the match to use are as follows:



The algebraic pullback approach (Sect. 4.2.5 on page 56) allows a direct representation of edNCE embedding, since a built-in feature of this approach is that a match of a left-hand side may associate an object of the left-hand side with an arbitrary number of objects of the host graph [8]. Again, only the sequential case was investigated. Since the pullback approach (and the extension, relation-algebraic rewriting) has some drawbacks with regard to a later implementation, we did not examine these techniques in detail.

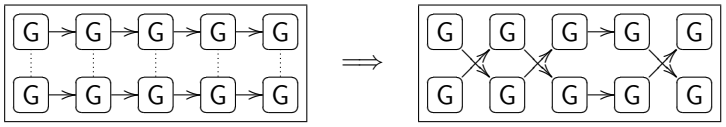
5.3.2 Productions with Gluing

Allowing only L-system-style productions would unnecessarily restrict the expressiveness of the formalism. As an example, they do not contain the possibility to keep existing nodes in the graph and to simply change edges between them or connect new nodes with them. This is the domain of gluing approaches like the algebraic approaches which establish connections by means of the kept nodes. Two examples for edge replacement are given by these productions in SPO notation:

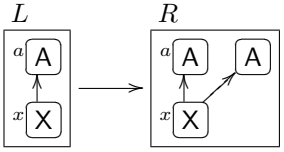


The first production moves a node of type A along an edge from an X-typed node to another X-typed node. This could be used to model the movement of, say, an animal a (type A) on a discretized world consisting of cells (nodes of type X) and neighbourhood edges. The second production models crossing-over of genomes, each genome being represented by a linear sequence of gene nodes of type G [98, 99]. The dotted line indicates an application condition, namely that the involved genes are on homologous gene loci.

Both productions can be applied as part of parallel derivations (Def. 4.49 on page 69) without any problems. This would model the independent (parallel) movement of animals on the discrete world, or the process of crossing-over at different locations at the same time. In both cases, nodes play the role of a fixed context for gluing, because they reappear on the right-hand sides. The following shows an example for a parallel derivation using the crossing-over production via three matches. (In practice, the choice whether to apply crossing-over at a possible match could be modelled by a stochastic process in order to reflect the situation in real life.)



Another case which is easily handled by (possibly parallel) SPO productions is the addition of nodes while keeping the rest. The following rule could model the creation of a child of animal a at the same cell x.



A quite large set of models can be built using only productions of this and the previous kind, i. e., SPO productions which keep some nodes of their left-hand side. This includes models based on a fixed, discrete world such as cellular automata or even board games like chess. As long as no nodes are deleted, we could equally well use the DPO approach since it is then equivalent with the SPO approach. However, if nodes are deleted, the DPO gluing condition may not always be satisfied. The resolution of these problematic cases within the SPO approach is reasonable for the purpose of modelling biological systems:

- The SPO approach removes dangling edges. Without this feature, it would not be possible to delete a node without knowing its complete neighbourhood in advance and specifying the latter as part of the left-hand side of the deleting production. The deletion of nodes is frequently required within models of the intended fields of applications, e. g., the dying off of branches of a plant, but usually we do not know the exact neighbourhood in advance: a dying branch may bear further branches or not, there may exist further edges depending on whether some further relationships with other nodes exist or not. The SPO solution to remove dangling edges seems useful for these important cases.
- The SPO approach removes nodes and edges which have at least one preimage with respect to the match that is deleted. The thus resolved conflicts between preservation and deletion do not only happen if a match for a single production identifies several left-hand side objects, of which only a subset is to be deleted, with the same object in the graph, they also happen for parallel productions where a subproduction deletes some object and another subproduction preserves the same object. Since in our considered models deletion is a deliberate and rather drastic event, is it reasonable to prefer deletion over preservation. For example, if there is a production for the metabolism of a plant organ, it will preserve the organ and simply update some state attributes. Another independent production could model the dying off of the organ by deletion, and its effect should not be inhibited by the existence of the metabolism production. Or in the case of animal movement, the movement production keeps the animal in the graph. There could be another production which deletes the animal because of insufficient energy or ageing. The mere existence of the movement rule should not protect the animal from being deleted.

5.3.3 Choice of Connection Mechanism: Single-Pushout Approach with Operators

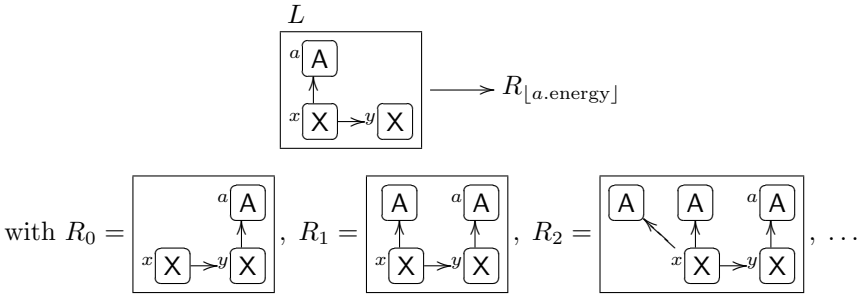
Based on the previous considerations, we chose a combination of the SPO approach with operators as a basis for relational growth grammars. Of course, this may be criticized from a theoretical point of view as a mixture of two very different approaches, but the power of both approaches is required for a

suitable formalism for functional-structural plant modelling and similar fields of application. The combination is intelligible and works very well in practice as we will see in Part III. Fortunately, it is possible to integrate the effect of operators within the SPO approach in a similar way as amalgamated two-level derivations can integrate embedding instructions in their first level (see page 64 and the discussion in Sect. 5.3.1 on page 102).

5.4 Dynamic Creation of Successor

With some exceptions, productions are given by static structures for left- and right-hand sides in the literature. This may be inconvenient in practice: think of a growth rule of a plant where the number of generated plant segments (i. e., nodes of the graph) depends on the local vitality of the plant. If we were constrained to conventional static productions, we would have to either specify several productions (one for each possible number of generated segments) or to use several derivation steps for a single global step of plant development (with the complicating possibility of different numbers of derivation steps at different locations of the plant). Both solutions are not feasible from a practical point of view. A better solution is to dynamically create the successor (i. e., the graph which replaces the match) on the basis of a match for the left-hand side. Within the context of L-systems, this was theoretically investigated in [23] where the successor of an L-system production is given by a mapping of the predecessor and its parameters, and it was practically implemented as the repetition and expansion operators for growth grammars (Sect. 3.11 on page 30), the dynamic construction mechanism of the L+C programming language (Sect. 3.14 on page 33), or the set of control statements of the ELSYS language [71].

The following is an example for such a production with dynamic successor on the basis of the SPO approach. It is a combination of the previous examples for animal movement and child creation where the number of created children depends on the energy (a node attribute). For the sake of simplicity, the reduction of energy on movement and the initial energy of children are omitted.



a.energy refers to the value of the energy attribute of the animal a, and $\lfloor x \rfloor$ denotes the integral part of x (floor function).

5.5 Rules

In this section, the main part of the RGG formalism is defined, namely its *rules*. We use the term rule instead of production for disambiguation purposes, since the application of an RGG rule will be defined via a dynamically created SPO production. The intent of the definitions of this section is to clarify the formal basis of relational growth grammars, we will not use them for proving theorems. In order to obtain (interesting) theoretical results, one might have to reduce the expressiveness until the definitions become well-suited for proving.

In the sequel, we use $\mathrm{Hom}(G, \cdot)$ to denote the set of all total homomorphisms in **RGGGraph** whose domain is G, and $\mathrm{Mon}_P(\cdot, \cdot)$ to denote the set of all partial monomorphisms, i. e., injective partial homomorphisms, in **RGGGraph**. If we simply write graph, an RGG graph is meant.

We start with the definition of an *application condition*. Given a potential match for the left hand side of a rule, such a condition determines whether the potential match is an actual match or not. Because every boolean expression of the concrete RGG language shall be allowed, we do not impose any restriction on the structure of a general application condition. We only demand decidability, i. e., the existence of an algorithm to decide whether the application condition is fulfilled or not for a given match, for obvious reasons.

Definition 5.11 (application condition). *Let L be a graph. An* application condition *c on* $\mathrm{Hom}(L, \cdot)$ *is a decidable subset of* $\mathrm{Hom}(L, \cdot)$*. It is* fulfilled *for $x : L \to X$ if $x \in c$.*

This definition follows [46]. More restricted application conditions in the context of the algebraic approach of graph transformation are discussed in [44].

Now we are ready to define a *rule* of a relational growth grammar. The definition is based on that of a typed attributed SPO production with inheritance in Def. 4.80 on page 79. However, a rule is not an SPO production, it rather generates such a production dynamically on the basis of a match. The match m of a rule also serves as the match for the generated production by letting the left-hand side of the production be a supergraph of the image $m(L)$ of the match. Right-hand side and homomorphism of the production are given by a mapping $p(m)$. This dynamic mechanism formally captures the dynamic generation of the successor as it has been discussed in the previous section. The fact that the left-hand side of the generated production may be a supergraph of $m(L)$ allows to match additional objects of the host graph in the production $p(m)$ which are not matched by the rule. However, we demand that these additional matches are kept as they are by the production so that

only matches of the original rule may be deleted. This means that additional matches have to be in the context, i. e., they have to be in the domain of the homomorphism of the production.

Furthermore, we integrate the operator approach by the inclusion of a mapping which assigns to each match a set of connection transformations. While the original definition (Def. 4.39 on page 66) is based on left-hand sides which consist of a single node, we extend the mechanism to arbitrary left-hand sides by letting a connection transformation be given by a 6-tuple $(s, (A, d, \gamma, h), t)$ where s is a node in the match of the left-hand side, t is a node of the right-hand side, A is an operator, $d \in \{\text{in}, \text{out}\}$ a direction flag, $\gamma \in \widehat{\Lambda}_E$ a concrete edge type, and $h \in \{0, 1\}$ determines if the connection transformation shall also create a connection edge if there is no matching connection transformation (see below in Def. 5.16). We may think of such a transformation as a *connection transformation edge* which points from the old node s of the host graph to the new node t. The mechanism of the operator approach then shifts connections along these edges from the host graph to the derived graph.

Definition 5.12 (RGG rule). *An* RGG rule $r = (L, c, p, z)$ *is given by a graph L, an application condition c, a mapping $p : \text{Hom}(L, \cdot) \to \text{Mon}_P(\cdot, \cdot)$ and a mapping z such that the following conditions hold:*

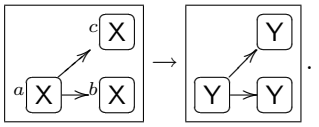
1. *For a match $m : L \to G \in \text{Hom}(L, \cdot)$ the image $p(m)$ is a typed attributed SPO production with inheritance $M(m) \xrightarrow{p(m)} R(m)$ whose domain (in the categorical sense) $M(m)$ is a subgraph of G and a supergraph of $m(L)$, $m(L) \sqsubseteq M(m) \sqsubseteq G$, and which is defined (as a graph monomorphism) for all objects not in $m(L)$, $M(m) \setminus m(L) \subseteq \text{dom}_P\, p(m)$.*
2. *For a match $m : L \to G \in \text{Hom}(L, \cdot)$ the image $z(m)$ is a finite set of connection transformations $(s, (A, d, \gamma, h), t)$ with an operator A, $s \in M(m), t \in R(m), d \in \{\text{in}, \text{out}\}, \gamma \in \widehat{\Lambda}_E, h \in \{0, 1\}$.*

Remark 5.13. An RGG rule may be just a conventional static SPO production $L \xrightarrow{x} R'$ if we set $p(m)$ to be the pushout arrow $m(L) \xrightarrow{x^*} R$ of the pushout of $m(L) \xleftarrow{m} L \xrightarrow{x} R'$.

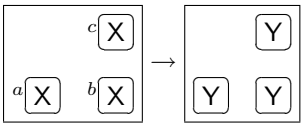
Remark 5.14. A production with operators $p : \mu \xrightarrow{\sigma,\tau} R$ has a natural translation to an RGG rule. The left-hand side L consists of a single μ-typed node n, we have no application condition, $p(m) : m(L) \to R$ has an empty domain (i. e., it is nowhere defined), and $z(m) = \bigcup_{(A,\gamma,w)\in\sigma}(m(n), (A, \text{out}, \gamma, 0), w) \cup \bigcup_{(A,\gamma,w)\in\tau}(m(n), (A, \text{in}, \gamma, 0), w)$.

Definition 5.15 (RGG match). *A* match *for a rule $r = (L, c, p, z)$ in a host graph G is a total graph homomorphism $m : L \to G \in \text{Hom}(L, \cdot)$ such that the application condition c is fulfilled.*

Given a set of rules with matches, the induced SPO productions can be applied in parallel, leading to a *parallel RGG derivation*. This is an important definition since it states how the parallelism of L-systems carries over in the RGG formalism. We also have to define how connection transformations are handled. For this purpose, we use a technique whose result is similar to an amalgamated two-level derivation, although it does not make use of amalgamation. If we reconsider the example of page 102, the final production (now as an SPO production) is



If we simply applied the original production p : [X] → [Y] via the three possible matches as part of a plain parallel SPO derivation, we would use the parallel production



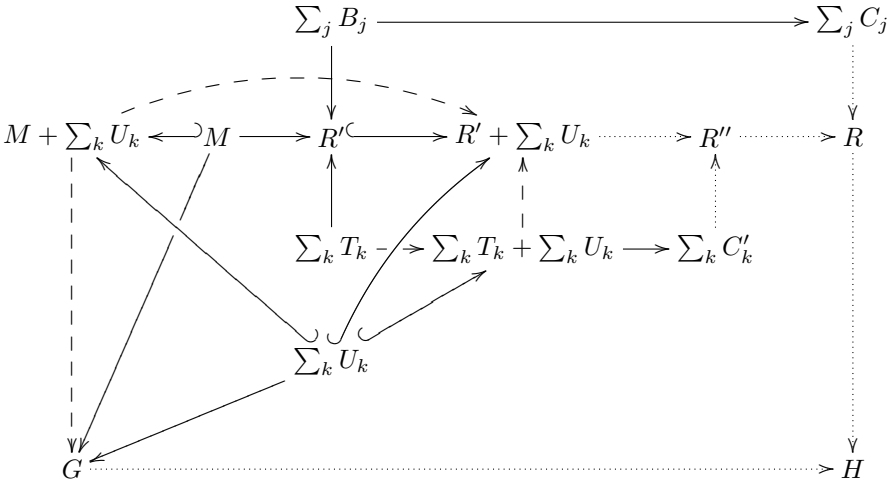
which does not create the desired connection edges. But obviously we can obtain the desired production by extending the right-hand side by connection edges (the extension of the left-hand side is not necessary because its nodes are deleted anyway). This mechanism can also be seen as a two-level derivation: At the first level, elementary SPO productions $p_i(m_i)$ are generated by matches m_i for rules r_i, combined to a parallel production and extended by connection edges according to the specified connection transformations $z_i(m_i)$. At the second level, a direct sequential derivation using the resulting production is executed. The difference to amalgamated two-level derivations is that the latter extend the elementary productions by connection edges at first and then amalgamate them to a parallel production.

Definition 5.16 (parallel RGG derivation). *Let G be a graph, I be a finite index set, $r = (r_i)_{i\in I}$ a family of rules with $r_i = (L_i, c_i, p_i, z_i)$, and $m = (m_i)_{i\in I}$ a family of finite sets of corresponding matches, i. e., every $f \in m_i$ is a match for r_i in the host graph G. A* direct parallel derivation *using r via m is given by a direct SPO derivation using $L \to R$ via the match $L \to G$, where $L \to R$ and $L \to G$ are determined as follows:*

1. *Let $M \to R'$ denote the coproduct $\sum_{i,f\in m_i} M_i(f) \xrightarrow{p_i(f)} R_i(f)$ of the productions. Let $Z = \bigcup_{i,f\in m_i} z_i(f)$ be the set which contains all connection transformations.*
2. *For each pair $(s, (A, d, \gamma, h), t), (s', (A', d', \gamma, h'), t') \in Z$ of matching connection transformations (i. e., $d \neq d', s' \in A_G(s), s \in A'_G(s')$), we construct the graph B which contains only t, t', the graph C which contains*

t, t' and a γ-typed edge between t and t' with t being the source if $d = \text{out}$ and the target otherwise. Let the families $(B_j)_{j\in J}, (C_j)_{j\in J}$ be chosen such that they exactly contain all such pairs of graphs B, C.

3. *For each connection transformation $(s, (A, d, \gamma, 1), t) \in Z$ and each node $s' \in A_G(s)$ which has no connection transformation for edge type γ and the opposite direction of d in Z, we construct the graph T which contains only t, the graph U which contains only s', the graph C' which contains t, s' and a γ-typed edge between t and s' with t being the source if $d = \text{out}$ and the target otherwise. Let the families $(T_k)_{k\in K}, (U_k)_{k\in K}, (C'_k)_{k\in K}$ be chosen such that they exactly contain all such triples of graphs T, U, C'.*
4. *We construct the following commutative diagram:*



The dashed arrows are uniquely determined by the universal property of the coproducts, the dotted arrows are pushouts. The graph L is given by $M + \sum_k U_k$ so that the diagram yields the homomorphisms $L \to R$, $L \to G$ and the direct SPO derivation $G \Rightarrow H$ using $L \to R$ via $L \to G$.

Remark 5.17. The construction seems to be quite complicated and is not very elegant. However, at least its effect is easy to understand. If there is a connection transformation $(s, (A, d, \gamma, 1), t)$ and a node $s' \in A_G(s)$ which has no connection transformation for edge type γ and the opposite direction of d, a connection edge between the new node t and the old node $s' \in G$ shall be created. (Note that this has no effect if s' is deleted by the derivation.) This new edge has to be established by the right-hand side which means that we have to include s' on the right-hand side and in the match. Thus we have to extend M to $M + \sum_k U_k$ and R' to $R' + \sum_k U_k$ and to include the new edges $\sum_k C'_k$ in R''. Finally we have to merge R'' with the new connection edges $\sum_j C_j$ resulting from matching pairs of connection transformations, this yields the final right-hand side R.

Remark 5.18. Item 3 of the definition extends the connection mechanism of operators by the convention that if for a node s' the set Z of all connection transformations has no connection transformation for a given edge type γ and the opposite direction of d, then connection edges shall be created for any connection transformation $(s, (A, d, \gamma, h), t)$ whose operator reaches s' and for which the h-component is set to 1. In other words, in such cases an implicit connection transformation from the node s' to itself is assumed whose operator reaches every node. This is similar to edNCE- or edNCEp-derivations (Def. 4.8 on page 47, Def. 4.45 on page 68) and is useful for cases where a node is kept by a derivation, while its neighbour is replaced by a new node:

$$G = \boxed{\boxed{\mathsf{A}} \to \boxed{\mathsf{B}}}, \quad \boxed{\mathsf{B}} \to \boxed{\mathsf{C}}\,.$$

Without the convention, the derived graph would not contain an edge from the A-node to the C-node even if the rule specifies a connection transformation for incoming edges of the B-node: there is no matching half-edge, i. e., no matching connection transformation associated with the A-node. But as this situation is very common (e. g., think of an internode A with a bud B which is replaced by a new organ C), we have to provide a "default connection transformation" from a node to itself. For the original definition of productions with operators, this situation cannot occur as every node is replaced in each derivation. By setting the h-component to 0, the default behaviour can be deactivated, and this is in fact advisable for translated L-system productions according to Def. 5.6 on page 100: the part $(N_\gamma^{\text{out}}, +, s)$ of σ should have $h = 0$ so that an edge whose source is replaced by a translated L-system production is not duplicated as a branch edge. All other parts of σ, τ should have $h = 1$ as motivated above.

Example 5.19. As an example for an RGG derivation, consider the following setting (where $*$ denotes the single edge type and N_*^d is given in Def. 5.6):

$$G = \boxed{\begin{array}{ccccc} & & & & {}^{d}\boxed{\mathsf{A}} \\ {}^{a}\boxed{\mathsf{A}} & \to & {}^{b}\boxed{\mathsf{B}} & \to & {}^{c}\boxed{\mathsf{B}} \end{array}},$$

$$r_1 = \Big(L_1 = {}^{\alpha}\boxed{\mathsf{A}}, \mathrm{Hom}(L_1, \cdot), m \mapsto \big(m(L_1) \to {}^{\alpha}\boxed{\mathsf{A}} \to {}^{\kappa}\boxed{\mathsf{C}}\big), m \mapsto \emptyset\Big),$$

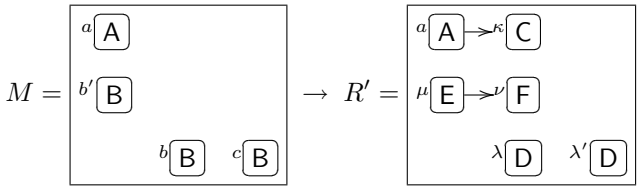
$$r_2 = \Big(L_2 = {}^{\beta}\boxed{\mathsf{B}}, \mathrm{Hom}(L_2, \cdot), m \mapsto \big(m(L_2) \to {}^{\lambda}\boxed{\mathsf{D}}\big),$$
$$m \mapsto \{(m(\beta), (N_*^{\text{in}}, \text{in}, *, 1), \lambda), (m(\beta), (N_*^{\text{out}}, \text{out}, *, 1), \lambda)\}\Big),$$

$$r_3 = \Big(L_3 = {}^{\gamma}\boxed{\mathsf{B}}, \mathrm{Hom}(L_3, \cdot), m \mapsto \big(m(L_3) \to {}^{\mu}\boxed{\mathsf{E}} \to {}^{\nu}\boxed{\mathsf{F}}\big),$$
$$m \mapsto \{(m(\gamma), (N_*^{\text{in}}, \text{in}, *, 1), \mu), (m(\gamma), (N_*^{\text{out}}, \text{out}, *, 1), \nu)\}\Big),$$

$$m_1 = \{(\alpha \mapsto a)\}\,,$$
$$m_2 = \{(\beta \mapsto b), (\beta \mapsto c)\}\,,$$
$$m_3 = \{(\gamma \mapsto b)\}\,.$$

The application conditions of all rules allow every match to be used, so actually the rules have no application condition. r_1 appends a C-typed node to an A-typed node, thereby keeping the latter. r_2 replaces a B-typed node by a D-typed node and moves both incoming and outgoing edges from the old node to the new node. r_3 replaces a B-typed node by a E-typed and an F-typed node, connected with an edge, and moves incoming edges to the E-typed node and outgoing edges to the F-typed node. r_1 is applied to the A-typed node a of G. r_2 is applied to both B-typed nodes b, c, while r_3 is only applied to b.

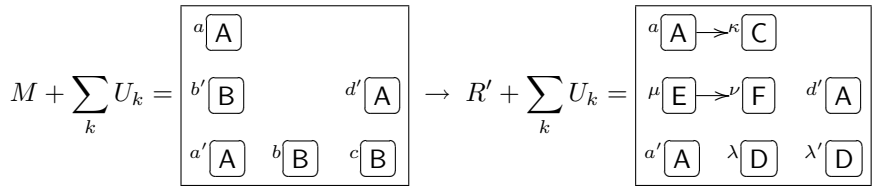
In the first step, the parallel production $M \to R'$ and the connection transformations Z resulting from all matches are computed:



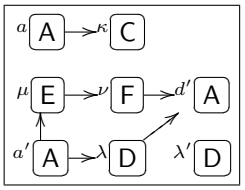
$$Z = \Big\{ \begin{array}{l} Z_1 = \big(b, (N_*^{\text{in}}, \text{in}, *, 1), \lambda\big), Z_2 = \big(b, (N_*^{\text{out}}, \text{out}, *, 1), \lambda\big), \\ Z_3 = \big(c, (N_*^{\text{in}}, \text{in}, *, 1), \lambda'\big), Z_4 = \big(c, (N_*^{\text{out}}, \text{out}, *, 1), \lambda'\big), \\ Z_5 = \big(b', (N_*^{\text{in}}, \text{in}, *, 1), \mu\big), Z_6 = \big(b', (N_*^{\text{out}}, \text{out}, *, 1), \nu\big) \Big\}. \end{array}$$

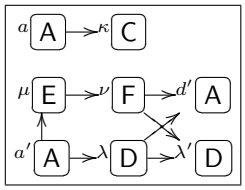
The mapping $M \to R'$ as well as the match $M \to G$ are indicated by the identifiers a, b, b', c, where both $b, b' \in M$ are mapped to $b \in G$.

Next, we have to consider all connection transformations in Z. Z_1 uses the edge from a to b to create an edge from a to λ, Z_2 uses the edge from b to d to create an edge from λ to d. Z_2 in combination with Z_3 detects the edge from b to c and creates an edge from λ to λ'. Z_4 does not lead to a connection edge. Z_5 is similar to Z_1 and creates an edge from a to μ. Finally, Z_6 is similar to Z_2 and creates both an edge from ν to d and in combination with Z_3 an edge from ν to λ'. Because a and d do not specify a connection transformation by themselves, they define the set $\sum_k U_k$ which has to be added to the production:



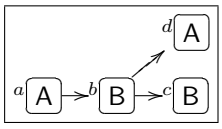
Both a, a' are mapped to $a \in G$, d, d' to $d \in G$. Now we can add the connection edges $\sum_k C'_k$ and $\sum_k C_k$ to the right-hand side:

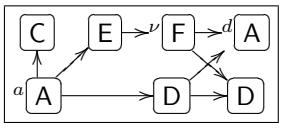
$R'' =$ 

, $R =$ 

.

Applied to G, this yields the parallel RGG derivation

$G =$ 

$\Longrightarrow$ $H =$ 

.

If, for example, the connection transformation of r_3 for outgoing edges has 0 instead of 1 as h-component, Z_6 changes to $(b', (N_*^{\text{out}}, \text{out}, *, 0), \nu)$. Then the edge from ν to d is not created as d has no matching connection transformation.

5.6 Control Flow and Relational Growth Grammar

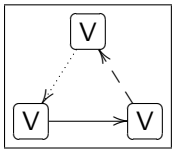
The simulation of the parallel mode of a (deterministic) L-system is obtained if every rule is applied via every possible match within a single parallel derivation. Generalizations like table L-systems [166] divide the set of productions into subsets such that, depending on the current state which may be, e. g., the number of already performed derivations or the current word, only productions from one subset are active within a derivation. Similar regulations of active productions have also been defined for graph grammars, a review is contained in [173]. All these mechanisms can be captured by a *control flow* which selects rules and their matches based on the current state which is taken here to be the host graph. (We could also model the state by some variable outside of the graph, which is more realistic in practice, but for the theory we may think of some distinguished state node in the graph with a lot of attributes.)

Definition 5.20 (control flow). *Let r be a family of rules as before. A* control flow *φ for r is a mapping which assigns to each graph G a family of finite (possibly empty) sets of matches for r in G, $\varphi : G \mapsto (m_i)_{i\in I}$ with $m_i \subseteq \text{Hom}(L_i, G)$. The derivation $G \overset{\varphi}{\Rightarrow} H$ according to the control flow φ is the parallel RGG derivation using r via $\varphi(G)$.*

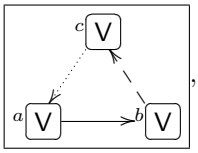
Definition 5.21 (relational growth grammar). *A* relational growth grammar *is given by a family r of rules with a control flow φ and a* start graph *α. The sequence of generated graphs G_n is given by derivations according to the control flow, $\alpha \left(\overset{\varphi}{\Rightarrow}\right)^n G_n$. The* generated language *is the set $\bigcup_{n\in\mathbb{N}_0} G_n$.*

Remark 5.22. The definition of a control flow and its derivation is deterministic, so relational growth grammars are deterministic. Because their field of application uses them to generate graphs, this is no practical restriction: the control flow may utilize a pseudorandom number generator to choose rules and matches; its seed would be part of the state, i. e., of the current graph.

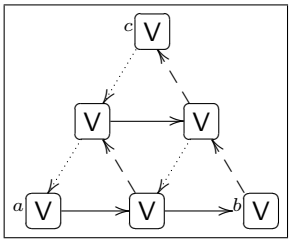
Example 5.23 (Sierpinski grammar). A simple example for a relational growth grammar is the Sierpinski grammar [191]. It models the Sierpinski construction (see Sect. 2.3 on page 13 and L-system (3.2) on page 21) as a graph, but not in the sense of Ex. 4.32 on page 61 where a node represents a black triangle: instead of this, a node represents a vertex so that each black triangle is represented as a graph of three circularly connected vertices. We use a node type V for the vertices and the three edge types e_0, e_{120}, e_{240} from the previous Sierpinski examples which stand for edges at 0°, 120° and 240°, respectively, in the usual 2D representation and which are drawn as solid, dashed, or dotted arrows. So the start graph (the initiator of the Sierpinski construction) is given by

$\alpha =$ 

and the rule (generator of the Sierpinski construction) is given by

$r = \Big(L =$ 

$, \mathrm{Hom}(L, \cdot),$

$m \mapsto \Big(m(L) \to$ 

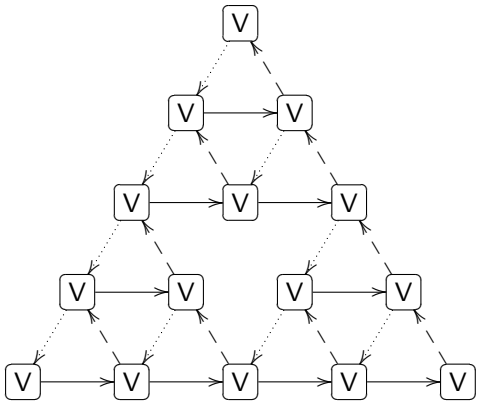
$\Big), m \mapsto \emptyset \Big).$

The usage of three edge types allows to identify "black" triangles, i. e., those triangles where the generator shall be applied. For example, the triangle in the centre of the right-hand side is not black because its edges are arranged in the wrong order. It does not match the left-hand side.

Now the control flow simply applies r in parallel at every possible location, i. e., it assigns to the current graph G all possible homomorphisms from L to G as matches for r:

$$\varphi : G \mapsto (\mathrm{Hom}(L, G)).$$

Then the generated language of this relational growth grammar consists of all finite approximations of the Sierpinski graph. After two derivation steps, we obtain

$G_2 =$  .

5.7 Relations within Rules

As the name indicates, relational notions play an important role within relational growth grammars and their applications. Relations in this context are binary predicates for nodes: given two nodes a, b, a relation R either exists between them or not. The simplest type of relations is given by edges. Edges "materially" or intrinsically represent the existence of a relation between their incident nodes. But we may also have derived relations which are given by some algorithmic prescriptions, e. g., spatial distance being below a threshold. This is similar to the distinction between intrinsic and derived attributes, where values of the former are directly stored as part of their owner, while the latter are given by formulas involving attributes and structure. The basic notion is captured by the definition of an RGG relation.

Definition 5.24 (RGG relation). *An* RGG relation R *is, for each graph* G*, a decidable subset* R_G *of* $G_V \times G_V$*. If* $(a, b) \in R_G$*, i. e., if the nodes* a, b *of* G *are in relation according to* R*, we also write* $R_G(a, b)$*.*

The special case of edges is obtained straightforwardly:

Definition 5.25 (edge relation). *An* edge relation R^e *is induced by an edge type* $e \in \Lambda_E$*, namely* $R^e_G(a, b) \Leftrightarrow \exists (a, t, b) \in G, t \leq_{\Lambda_E} e$*.*

Also boolean-valued functions (binary predicates) and unary node-valued functions induce relations in a natural way:

Definition 5.26 (binary predicate relation). *A* binary predicate relation R^p *is induced by a* G*-indexed family of boolean-valued functions* $p_G : G_V \times G_V \to \{0, 1\}$*, namely* $R^p_G(a, b) \Leftrightarrow p_G(a, b) = 1$*.*

Definition 5.27 (node function relation). *A* node function relation R^f *is induced by a G-indexed family of unary node-valued functions $f_G : G_V \to G_V$, namely $R^f_G(a,b) \Leftrightarrow b = f_G(a)$.*

Given a relation R, new relations can be defined on its basis. The most important ones are *transitive closures*. If $R_G(a,b)$ and $R_G(b,c)$, this does not necessarily imply $R_G(a,c)$, i. e., R_G is not necessarily transitive. However, this transitivity holds for the transitive closure of R which is the least transitive relation which contains R as a subrelation, i. e., which relates all elements related by R. As an example, the transitive closure of an edge relation R^e relates an ordered pair (a,b) of nodes if they are connected by a path of e-typed edges from a to b. If the transitive closure is reflexive, nodes are in relation with themselves, i. e., even paths of zero length are considered.

Definition 5.28 (transitive closure). *Let R be an RGG relation. Its* transitive closure R^+ *is, for every graph G, the least transitive relation R^+_G with $R_G(a,b) \Rightarrow R^+_G(a,b)$. Its* reflexive, transitive closure R^* *is, for every graph G, the least reflexive, transitive relation R^*_G with $R_G(a,b) \Rightarrow R^*_G(a,b)$.*

As relations are a generalization of edges, it makes perfectly sense to use them for the left-hand side of a rule (L,c,p,z) just like edges. In the theory, edges of the left-hand side can be specified directly as part of L, while general relations have to be included as part of the application condition c. A set of relations is thus treated as a *relational application condition* which is fulfilled if and only if every single relation is fulfilled, where a single relation R^i is evaluated for the match of two nodes a_i, b_i of the left-hand side of a rule.

Definition 5.29 (relational application condition). *A* relational application condition c *on* $\mathrm{Hom}(L,\cdot)$ *is an application condition of the form $c = \left\{ x \in \mathrm{Hom}(L,\cdot) \,\middle|\, \forall i \in I : R^i_{\mathrm{cod}\,x}(x(a_i), x(b_i)) \right\}$ with a finite index set I, relations R^i and nodes $a_i, b_i \in L$.*

Note that although the theory makes a distinction between edges and general relations, a concrete programming language may provide a syntax for left-hand sides of rules which unifies edges and general relations (see Sect. 6.5.3 on page 148).

5.8 Incremental Modification of Attribute Values

Parallel SPO derivations are well-defined if they modify the structure and there are no graph constraints restricting the allowed number of edges at a node. Conflicts are resolved in a unique and meaningful manner. However, parallel modifications of attribute values may lead to more than one attribute edge for the same attribute at a single node, this of course violates the graph constraints for attributed graphs (the graph is not strict in the sense

of Def. 4.71 on page 77). There is simply no sensible definition of the assignment of different values to an attribute of an object in parallel. What does make sense is the *incremental modification* of attribute values in parallel: the addition of several increments, the multiplication by several factors, or in general the application of mutually commutative operations in parallel. We may think of some dynamical system with independent processes (for example, a time-discrete approximation of a system of ordinary differential equations), each of which depends on numeric attributes of nodes and at the same time modifies these attributes by small increments within a step. In order to implement such a system using an imperative programming language, one usually makes a copy of the current state, uses this copy throughout the whole step as input to the processes and then computes and applies the increments in a sequential way. By using the copy, one ensures that the processes consistently use the previous values of attributes instead of a mixture of previous, intermediate and new ones depending on the order of sequentialization. (But there are also algorithms like the Gauß-Seidel method which take advantage of the early availability of at least some new values.)

Within parallel graph rewriting, we can (and have to) take another way: we can use *incremental modification edges* from the node whose attribute is to be modified to the value by which it is modified. After a parallel derivation which has created such edges, the control flow has to apply *cumulating productions* sequentially as long as possible, where each sequential derivation cumulates a single incremental modification edge into the attribute value as shown in Fig. 5.1. Of course, we have to impose the graph constraint that for each object and attribute there are at most incremental modification edges of mutually commutative operations. E. g., incremental addition and multiplication at the same time are not commutative and, thus, cannot be defined reasonably.

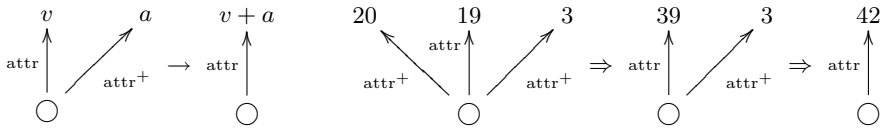


Figure 5.1. Incremental modification: (**a**) cumulating production for incremental addition of attribute 'attr', additive increments are encoded by edge type 'attr$^+$'; (**b**) sequential derivations perform addition

Appendix 5.A Proofs

In order to prove Theorem 5.8, we need an auxiliary lemma concerning the translation T from well-nested words to graphs (see Def. 5.2 on page 96):

Lemma 5.30 (decomposition law for T). *Let χ, ν be well-nested words over V. For node sets R, L, B, where L, R may also contain the non-node symbol %, define*

$$E(R,L,B) := \bigcup_{n\in R\setminus\{\%\}} \left(\bigcup_{m\in L\setminus\{\%\}} (n,>,m) \cup \bigcup_{m\in B} (n,+,m) \right) .$$

I. e., the set $E(R,L,B)$ contains exactly those edges which arise when the nodes of R are connected with the nodes of L by successor edges and with the nodes of B by branch edges. Then we have

$$\begin{aligned}
T^G(\chi\nu) &= T^G(\chi) \cup T^G(\nu) \cup E\left(T^R(\chi), T^L(\nu), T^B(\nu)\right) , \\
T^L(\chi\nu) &= \begin{cases} T^L(\chi) : T^L(\chi) \neq \emptyset \\ T^L(\nu) : \textit{otherwise} \end{cases} , \\
T^B(\chi\nu) &= \begin{cases} T^B(\chi) : T^L(\chi) \neq \emptyset \\ T^B(\chi) \cup T^B(\nu) : \textit{otherwise} \end{cases} .
\end{aligned}$$

Remark 5.31. To simplify the notation, we assume that the node n to which a symbol a at a given position p in the current word μ is mapped by T is independent of the splitting of μ, i. e., if $\chi\nu = \chi'\nu' = \mu$ are two splittings of μ such that p is contained in χ and ν', the corresponding node in $T^G(\chi\nu), T^G(\chi'\nu')$ is the same. Without this simplification, we would have to work with equivalence classes of graphs.

Proof. The decomposition law for T^L and T^B is obtained easily: if $T^L(\chi) \neq \emptyset$, it follows from Def. 5.2 on page 96 that there must be some symbol $a \in V \cup \{\%\}$ such that $\chi = \mu a \omega$ where ω is a well-nested word and μ is a possibly empty sequence of well-nested words in brackets, $\mu = [\mu_1] \dots [\mu_k]$. But then a is also the left-most non-bracketed symbol of $T^L(\chi\nu)$, and it follows from Def. 5.2 that $T^L(\chi\nu) = T^L(\chi)$ and $T^B(\chi\nu) = T^B(\chi)$. Otherwise, if $T^L(\chi) = \emptyset$, it follows $\chi = [\mu_1] \dots [\mu_k]$ and then $T^L(\chi\nu) = T^L(\nu), T^B(\chi\nu) = T^B(\chi) \cup T^B(\nu)$.

The identity for T^G is proved by structural induction with respect to χ according to the grammar for well-nested words in Def. 5.1 on page 96. The induction starts with the empty word $\chi = \varepsilon$ for which the identities are easily verified. Assuming that we have proved the claim for χ, we have to show its validity for $\%\chi$, $[\chi']\chi$ and $a\chi$. For the first case, we have

$$\begin{aligned}
T^G(\%\chi\nu) = T^G(\chi\nu) &= T^G(\chi) \cup T^G(\nu) \cup E\left(T^R(\chi), T^L(\nu), T^B(\nu)\right) \\
&= T^G(\chi) \cup T^G(\nu) \cup E\left(\{\%\} \cup T^R(\chi), T^L(\nu), T^B(\nu)\right) \\
&= T^G(\%\chi) \cup T^G(\nu) \cup E\left(T^R(\%\chi), T^L(\nu), T^B(\nu)\right) .
\end{aligned}$$

For the second case, we have

$$\begin{aligned}
T^G([\chi']\chi\nu) &= T^G(\chi') \cup T^G(\chi\nu) \\
&= T^G(\chi') \cup T^G(\chi) \cup T^G(\nu) \cup E\left(T^R(\chi), T^L(\nu), T^B(\nu)\right) \\
&= T^G([\chi']\chi) \cup T^G(\nu) \cup E\left(T^R([\chi']\chi), T^L(\nu), T^B(\nu)\right) .
\end{aligned}$$

For the third case, we have to show that

$$\begin{aligned} T^G(a\chi\nu) &= \{n\} \cup T^G(\chi\nu) \cup E\left(\{n\}, T^L(\chi\nu), T^B(\chi\nu)\right) \\ &= \{n\} \cup T^G(\chi) \cup T^G(\nu) \cup E\left(T^R(\chi), T^L(\nu), T^B(\nu)\right) \\ &\quad \cup E\left(\{n\}, T^L(\chi\nu), T^B(\chi\nu)\right) \end{aligned}$$

equals

$$\begin{aligned} T^G(a\chi) \cup T^G(\nu) \cup E\left(T^R(a\chi), T^L(\nu), T^B(\nu)\right) \\ = \{n\} \cup T^G(\chi) \cup T^G(\nu) \cup E\left(T^R(a\chi), T^L(\nu), T^B(\nu)\right) \\ \cup E\left(\{n\}, T^L(\chi), T^B(\chi)\right) \ . \end{aligned}$$

If $T^L(\chi) \neq \emptyset$, we also have $T^R(\chi) \neq \emptyset$ and conclude the equality from Def. 5.2 on page 96 and this lemma for T^L, T^B. Otherwise, If $T^L(\chi) = \emptyset$, we also have $T^R(\chi) = \emptyset$ and conclude the equality in a similar way. □

Now we are ready to prove the equivalence theorem from page 100, which is repeated here:

Theorem 5.8 (equivalence of axis-propagating L-system and translated graph grammar). *Let V, V_B be as in Def. 5.1 on page 96. Let $\mathcal{G} = (V_B, \alpha, P)$ be a D0L-system such that $P = \{\% \to \%, [\to [,] \to]\} \cup P'$ where P' contains only axis-propagating productions. Then $\mathcal{G}$ is equivalent to the translated parallel graph grammar with operators $T(\mathcal{G}) = \left(T^G(\alpha), \{T(p) \,|\, p \in P'\}\right)$ in the sense that for every derivation the diagram*

$$\begin{array}{ccc} \mu & \xRightarrow{\mathcal{G}} & \nu \\ {\scriptstyle T^G}\downarrow & & \downarrow{\scriptstyle T^G} \\ T^G(\mu) & \xRightarrow{T(\mathcal{G})} & T^G(\nu) \end{array}$$

commutes.

Proof. We prove the theorem by structural induction with respect to μ according to the grammar for well-nested words in Def. 5.1 on page 96. In order for this to work, we have to prove five additional properties:

1. For $t \in T^L(\nu) \backslash \{\%\}$, there is a node $u \in T^L(\mu)$ with connection transformations $(N^{\text{in}}_{>}, >, t), (N^{\text{in}}_{+}, +, t) \in \tau^u$.
2. For $t \in T^B(\nu)$, there is a node $u \in T^L(\mu)$ with connection transformations $(N^{\text{in}}_{>}, +, t), (N^{\text{in}}_{+}, +, t) \in \tau^u$ or a node $u \in T^B(\mu)$ with a connection transformation $(N^{\text{in}}_{+}, +, t) \in \tau^u$.
3. For $u \in T^L(\mu) \backslash \{\%\}$ and $(N^{\text{in}}_{>}, \gamma, t) \in \tau^u$, we have either $t \in T^L(\nu)$ and $\gamma = >$ or $t \in T^B(\nu)$ and $\gamma = +$.

4. For $u \in T^L(\mu)\backslash\{\%\}$ and $(N_+^{\text{in}}, \gamma, t) \in \tau^u$, we have $t \in T^L(\nu) \cup T^B(\nu)$ and $\gamma = +$.
5. For $u \in T^B(\mu)$ and $(N_+^{\text{in}}, \gamma, t) \in \tau^u$, we have $t \in T^B(\nu)$ and $\gamma = +$.

τ^u denotes the set of connection transformations which are associated with the predecessor node u by the derivation within $T(\mathcal{G})$, see the notation of Def. 4.41 on page 66. But note that we simplify the notation with respect to t: t is a node of the derived graph, while connection transformations use nodes of productions. With the notation of Def. 4.41 we would have to write $h^{-1}(t)$ instead of t.

The induction starts with the empty word $\mu = \varepsilon$ for which all conditions are trivially satisfied. Assuming that we have proved the cases $\mu \overset{\mathcal{G}}{\Rightarrow} \nu$ and $\mu' \overset{\mathcal{G}}{\Rightarrow} \nu'$, the diagram

$$\begin{array}{ccc} \%\mu & \overset{\mathcal{G}}{\Longrightarrow} & \%\nu \\ {\scriptstyle T^G}\downarrow & & \downarrow{\scriptstyle T^G} \\ T^G(\%\mu) & \overset{T(\mathcal{G})}{\Longrightarrow} & T^G(\%\nu) \end{array}$$

commutes because $T^G(\%\mu) = T^G(\mu), T^G(\%\nu) = T^G(\nu)$, and the additional properties hold because the sets are empty. (For $\%\mu \overset{\mathcal{G}}{\Rightarrow} \%\nu$ we have used the fact that $\mathcal{G}$ replaces % by %.) Also the diagram

$$\begin{array}{ccc} [\mu']\mu & \overset{\mathcal{G}}{\Longrightarrow} & [\nu']\nu \\ {\scriptstyle T^G}\downarrow & & \downarrow{\scriptstyle T^G} \\ T^G([\mu']\mu) & \overset{T(\mathcal{G})}{\Longrightarrow} & T^G([\nu']\nu) \end{array}$$

commutes: we have $T^G([\mu']\mu) = T^G(\mu') \cup T^G(\mu)$ so that there are no edges between the parts $T^G(\mu')$ and $T^G(\mu)$. Consequently, there are no connection edges between the successors of these parts because the operators N_λ^d cannot reach from one part to the other. So the derivation within $T(\mathcal{G})$ can be done separately for each part, this yields commutativity for the diagram. Since $T^L([\mu']\mu) = T^L(\mu)$, $T^L([\nu']\nu) = T^L(\nu)$ and $T^B([\nu']\nu) \supseteq T^B(\nu)$, the validity of properties 1, 3 and 4 follows directly from the validity for the case $\mu \overset{\mathcal{G}}{\Rightarrow} \nu$. For property 2 with $t \in T^B([\nu']\nu)$, we

- either have $t \in T^B(\nu)$ and the validity follows from the same property for $\mu \overset{\mathcal{G}}{\Rightarrow} \nu$,
- or we have $t \in T^B(\nu')$ and conclude from property 2 of $\mu' \overset{\mathcal{G}}{\Rightarrow} \nu'$ that there is a node $u \in T^L(\mu') \cup T^B(\mu') \subseteq T^B([\mu']\mu)$ with $(N_+^{\text{in}}, +, t) \in \tau^u$,
- or we have $t \in T^L(\nu')$ and conclude from property 1 of $\mu' \overset{\mathcal{G}}{\Rightarrow} \nu'$ that there is a node $u \in T^L(\mu') \subseteq T^B([\mu']\mu)$ with $(N_+^{\text{in}}, +, t) \in \tau^u$.

For property 5 with $u \in T^B([\mu']\mu)$, we

- either have $u \in T^B(\mu)$ and use property 5 of $\mu \overset{\mathcal{G}}{\Rightarrow} \nu$,
- or we have $u \in T^B(\mu')$ and use property 5 of $\mu' \overset{\mathcal{G}}{\Rightarrow} \nu'$,
- or we have $u \in T^L(\mu')$ and use property 4 of $\mu' \overset{\mathcal{G}}{\Rightarrow} \nu'$ together with $T^L(\nu') \cup T^B(\nu') \subseteq T^B([\nu']\nu)$.

It remains to prove the case

$$\begin{array}{ccc} a\mu & \xRightarrow{\mathcal{G}} & \chi\nu \\ {\scriptstyle T^G}\downarrow & & \downarrow{\scriptstyle T^G} \\ T^G(a\mu) & \xRightarrow{T(\mathcal{G})} & T^G(\chi\nu) \end{array}$$

for $a \to \chi \in P, a \in V$. We have $T^G(a\mu) = \{n\} \cup T^G(\mu) \cup E(\{n\}, T^L(\mu), T^B(\mu))$ with some unique a-labelled node n. Lemma 5.30 allows us to decompose $T^G(\chi\nu) = T^G(\chi) \cup T^G(\nu) \cup E(T^R(\chi), T^L(\nu), T^B(\nu))$, Furthermore, we have $\{n\} \overset{T(\mathcal{G})}{\Longrightarrow} T^G(\chi)$ and $T^G(\mu) \overset{T(\mathcal{G})}{\Longrightarrow} T^G(\nu)$. What we have to show is that the connection edges between $T^G(\chi)$ and $T^G(\nu)$ which are created by the derivation of $T^G(a\mu)$ are exactly the edges $E(T^R(\chi), T^L(\nu), T^B(\nu))$.

- For an edge $(s, >, t)$ of the latter set, we have $s \in T^R(\chi), t \in T^L(\nu)$ and deduce from property 1 that there is a node $u \in T^L(\mu)$ with $(N^{\text{in}}_{>}, >, t) \in \tau^u$ and from Def. 5.6 on page 100 that $(N^{\text{out}}_{>}, >, s) \in \sigma^n$. As there is a successor edge from n to u in $T^G(a\mu)$, the operators match for (s, t) and thus create an edge $(s, >, t)$.
- For an edge $(s, +, t)$ we have $s \in T^R(\chi), t \in T^B(\nu)$ and deduce from property 2 that there is
  - a node $u \in T^L(\mu)$ with $(N^{\text{in}}_{>}, +, t) \in \tau^u$ and from Def. 5.6 that $(N^{\text{out}}_{>}, +, s) \in \sigma^n$. As there is a successor edge from n to u in $T^G(a\mu)$, the operators match for (s, t) and thus create an edge $(s, +, t)$.
  - or a node $u \in T^B(\mu)$ with $(N^{\text{in}}_{+}, +, t) \in \tau^u$ and from Def. 5.6 that $(N^{\text{out}}_{+}, +, s) \in \sigma^n$. As there is a branch edge from n to u in $T^G(a\mu)$, the operators match for (s, t) and thus create an edge $(s, +, t)$.

On the other hand, let (s, γ, t) be a connection edge created by $T(\mathcal{G})$ between the parts $T^G(\chi)$ and $T^G(\nu)$. Then there must be a predecessor node $u \in T^G(\mu)$ and an edge $(n, \lambda, u) \in T^G(a\mu)$ such that $(N^{\text{out}}_{\lambda}, \gamma, s) \in \sigma^n$ and $(N^{\text{in}}_{\lambda}, \gamma, t) \in \tau^u$. From $(N^{\text{out}}_{\lambda}, \gamma, s) \in \sigma^n$ and Def. 5.6 we deduce $s \in T^R(\chi)$. The existence of the edge (n, λ, u) implies that

- either $u \in T^L(\mu)$ and $\lambda = >$. In this case property 3 implies $t \in T^L(\nu)$ and $\gamma = >$ or $t \in T^B(\nu)$ and $\gamma = +$. But then (s, γ, t) is in $E(T^R(\chi), T^L(\nu), T^B(\nu))$.
- or $u \in T^B(\mu)$ and $\lambda = +$. From property 5 we deduce $t \in T^B(\nu)$ and $\gamma = +$ so that (s, γ, t) already is in $E(T^R(\chi), T^L(\nu), T^B(\nu))$.

Finally, we have to check the properties. We have $T^L(a\mu) = \{n\}$ and, due to the axis-propagating precondition $T^L(\chi) \neq \emptyset$ and Lemma 5.30, $T^L(\chi\nu) = T^L(\chi)$ and $T^B(\chi\nu) = T^B(\chi)$. Thus, for $t \in T^L(\chi\nu)$ we obtain $t \in T^L(\chi)$, and property 1 follows from Def. 5.6 with $u = n$. For $t \in T^B(\chi\nu)$ we obtain $t \in T^B(\chi)$, and property 2 follows analogously to property 1. For properties 3 and 4, we have $u = n$ again and use Def. 5.6 to show their validity. Property 5 is trivially fulfilled since $T^B(a\mu) = \emptyset$. □

Part II

Design and Implementation of the XL Programming Language

The concept of relational growth grammars which was introduced in Chap. 5 may serve as an underlying formalism for programming languages within the rule-based paradigm. One needs a concrete programming language which implements this concept in order to actually make use of relational growth grammars. Three tasks have to be coped with:

- A specification for the programming language has to be developed.
- An interpreter or compiler has to be implemented so that programs can be understood by a computer.
- A run-time library has to be provided which contains fundamental data structures, basic functions and, depending on the field of application, a set of further facilities, for example 3D geometric objects for 3D plant modelling and corresponding functions to compute geometric properties like the distance.

These tasks are considerably simplified if one uses an existing programming language as basis and builds an implementation language of relational growths grammars on top.

We chose the Java programming language as basis and defined the *XL programming language* as an extension of the Java programming language. For the XL programming language, the concept of relational growth grammars can be implemented in a relatively easy way. The name XL may be seen as a reminiscence of L-systems, namely as the acronym of eXtended L-systems.

Choosing the Java programming language as basis has several advantages:

- The Java programming language is a widely used, platform-independent object-oriented programming language.
- It has a clean and simple language design (simple at least compared to C++).
- Due to its popularity, the range of available libraries is one of the widest for programming languages. Even the standard run-time library covers a wide range of applications.
- Programs and libraries written in the Java programming language are usually compiled for the Java virtual machine. Compiling for the Java virtual machine is much simpler than compiling to typical native machine code.

Since the Java programming language in itself has no graph- or rule-based features, these requirements for relational growth grammars have to be addressed by the extension XL.

Of course, we could also have chosen an existing rule-based language as basis and enrich it with the missing features. However, existing rule-based languages and their libraries are not even approximately as widespread and extensive as it it the case for the Java programming language, so the increase in value inherited by the basis language would have been relatively small.

6

Design of the Language

This chapter describes the design of the XL programming language. Although the presentation is relatively streamlined, the process of finding this final design was not, but we believe that we arrived at a consistent, useful and general programming language not only for rule-based programming.

6.1 Requirements

The requirements which the XL programming language has to fulfil are derived from the possibility to specify relational growth grammars in a convenient way. They can be divided into several parts:

- A type graph with inheritance can be specified, the user can define new types.
- There is a syntax for rules in general, consisting of a left-hand side and a right-hand side (Def. 5.12 on page 107).
- Both SPO gluing (i. e., identification of objects of the left- and right-hand side) and connection transformations are supported.
- The rule syntax supports application conditions (Def. 5.11 on page 106).
- Relations as discussed in Sect. 5.7 on page 114 have a convenient syntax, including their transitive closures.
- The dynamic creation of the successor can be specified (Sect. 5.4 on page 105).
- Attribute values can be assigned as part of a parallel derivation. This includes the incremental modification of attribute values (Sect. 5.8 on page 115).
- The control flow which governs the application of rules is specifiable (see Def. 5.20 on page 112 and the discussion in Sect. 5.1 on page 92).
- Query and aggregate expressions are supported (see the discussion in Sect. 5.1 on page 93). Query expressions should reuse the syntax of left-hand sides.

6.2 Design Guidelines

Since the XL programming language shall be defined as an extension of the Java programming language, the extension has to be designed such that it coheres syntactically and semantically with the Java programming language. For example, the Java programming language makes extensive use of symbols, thus following the tradition of the C programming language: With the exception of **new** and **instanceof**, all operators are represented by symbols, subexpressions are tied together with parentheses. Statements are grouped together by blocks, whose boundaries are marked by the symbols { and }. Keywords are mainly used at higher levels, namely for statements which influence the control flow (e. g., **if**, **for**, **break**, **throw**), for declarations (e.g, **class**, **public**) and for primitive types. Thus, extensions of the Java programming language should use mostly symbols at the level of expressions, statements and grouping, and should use keywords at higher levels.

Besides coherence, also generality is one of the guidelines for the design of the XL programming language. The requirements posed by relational growth grammars are of a relatively concrete nature, but if one finds the "right" abstraction so that the XL programming language in itself makes no reference to a specific graph structure or a specific mode of rewriting at all, then it is possible to use the language not only for relational growth grammars, but also as a general language with support for the rule-based paradigm in various fields of application, e. g., the rule-based transformation of XML documents or a mechanism similar to vertex-vertex algebras to transform polygon meshes (Sect. 4.8.4 on page 85). Even the implementation of relational growth grammars presented in Appendix B derives benefit from this generality, as it provides for easily realizable extensions to the original specification of relational growth grammars. An obvious presupposition for generality is that the XL programming language must not make any reference to some target application, e. g., to the modelling platform GroIMP which is presented in Appendix A.

The underlying Java programming language does not define a syntax for rules, so in principle, we are free in designing this syntax. However, in this case the guideline of coherence should be followed with suitable existing rule-based languages in mind. We chose the L-system languages of GROGRA and cpfg as patterns.

In the following sections, we discuss the design of the new features of the XL programming language at which we finally arrived on the basis of the general guidelines. A thorough specification is presented in [95]. For a complete understanding, the reader is expected to be familiar with the Java programming language as defined by the Java language specification in its third edition [72]. The discussion often refers to types declared in subpackages of `de.grogra.xl`, additional Javadoc documentation for these types is available at the website of the project [101].

Note that the presented features are arranged in a logical order such that subsequent sections base on previous sections. For this reason, we do not start with the discussion of rules. However, for a more concrete discussion it is sometimes necessary to show code examples which use features that have not yet been explained.

6.3 Generator Expressions

One of the requirements for the XL programming language is to be able to use query expressions and aggregate functions. As we have discussed in Sect. 5.1 on page 93, query expressions find matches of a pattern in a graph such as all nodes of a given type which fulfil some condition, and aggregate functions compute aggregated values based on the result of a query, for example the sum of some numeric attribute of the found matches. Within the XL programming language, an example is the expression

```
sum((* f:F, (f.diameter > 0.01) *).length)
```

which finds all nodes of type `F` (think of nodes of a 3D scene graph representing a line or cylinder like the turtle command `F`), binds them to the identifier `f`, checks if the application condition `f.diameter > 0.01` is fulfilled, obtains their `length` values, and computes the sum of the latter. The query is the part enclosed in asterisked parentheses, `sum` is an aggregate function.

If we abstract from the example, we arrive at the classical producer/consumer pattern: a query expression is s special kind of a general class of expressions which produce multiple values, and an aggregate expression consumes multiple values as input and returns a single result. We call expressions which produce multiple values *generator expressions*, this name has been adopted from the Python programming language [162]. For the implementation of generator expressions, there are several possibilities:

1. We can return an array which contains all values. This would be a very simple solution which is already available within the Java programming language. But it has the major disadvantage that all values have to be computed and stored in advance. This is a disadvantage with respect to both time and memory. For example, assume that we want to know if the multiple values contain a value which fulfils some condition. If we have found such a value, we do no longer need the remaining ones. However, using an array they would have been computed in advance for nothing, thereby wasting time. And if we need multiple values just to compute an aggregated value, we often do not need simultaneous knowledge about all values. For example, to compute a sum it suffices to know a single value at a time and add this value to a temporary variable.
2. Likewise, we may use a `java.util.List`. Because lists allow random access to their elements, we also would have to compute the values in advance so that the same considerations as for arrays hold.

3. We may use a `java.util.Iterator`. This interface declares the method `hasNext` to check whether there is a further value and the method `next` to fetch this value. Using this technique, only those values are computed which are actually used, and there is no need to store all computed values simultaneously. This solves the problems concerning time and memory, thus iterators are a good candidate for generator expressions.
4. We can also use a callback interface with a single method `consume` which receives a value. Compared to the previous technique, this reverses the responsibilities: using iterators, the consumer which is interested in values invokes `next` on the producer to fetch the next value; using callback interfaces, the producer invokes `consume` on the consumer to pass the next value. This approach also has no memory overhead, and if the consumer has a possibility to tell the producer that no further values are needed, there is no unneeded computation of values. It is called *internal iterators* because the iteration logic is internal to the iterator, while the iterators of the previous approach are called *external* [62].
5. If the Java virtual machine were not the target machine, we could use coroutines [102]. Coroutines may temporarily suspend execution and give control back to their callers. Later, their execution can be resumed just after the point where they have been suspended. However, such a mechanism is in conflict with the purely stack-based Java virtual machine.

Only the options 3 and 4 are eligible. Using internal iterators, producers can be specified in a natural way. For example, with a suitable interface `Consumer` a method which produces all **int**-values from 0 to `n` is easily specified in the Java programming language like

```
void produce(Consumer<Integer> consumer, int n) {
    for (int i = 0; i <= n; i++) {
        consumer.consume(i);
    }
}
```

But the usage of this producer method is relatively complicated because we have to provide an implementation of the callback interface (a *closure*):

```
class Sum implements Consumer<Integer> {
    int tmp = 0;

    public void consume(Integer value) {
        tmp += value;
    }
}

Sum s = new Sum();
produce(s, 100);
int sum = s.tmp;
```

Note that although this example has no explicit support to allow the consumer to terminate the producer loop, this is nevertheless possible by throwing and catching an exception.

Using external iterators, the situation is reversed: external iterators require to remember the state of the producer between invocations of `next` which makes producer implementations difficult. The producer could be implemented like

```
class Producer implements Iterator<Integer> {
    int i = 0;
    int n;
    boolean done;

    Producer(int n) {
        this.n = n;
        this.done = n < 0;
    }

    public boolean hasNext() {
        return !done;
    }

    public Integer next() {
        done = i == n;
        return i++;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

In this case, remembering the state is simple. But for more complex producers, the state becomes complex and even includes several stack frames for recursive algorithms like pattern matchers for queries. Because the Java virtual machine does only allow to access its current stack frame, recursive algorithms would have to be specified without recursive method invocations, i. e., they would have to manage a stack-like structure (e. g., a tree path) by their own. On the other hand, external iterators allow the consumer to be implemented in a natural way:

```
int sum = 0;
for (Iterator<Integer> it = new Producer(100); it.hasNext(); ) {
    sum += it.next();
}
```

A technique to convert an internal iterator to an external one is to use an own thread for the internal iterator and to properly synchronize the methods `consume` and `next` (which run on different threads). This is the classical solution for the producer/consumer pattern and is also used by pipes of modern

operation systems. However, it is only appropriate for transferring large data streams because setting up an own thread and the synchronization mechanism require a lot of time. Thus, it is not feasible for our purposes since query expressions may be evaluated frequently and typically return only small sets of values.

The conversion from an external iterator to an internal one is simple, we just need a loop which iterates over the external iterator and passes every value to the consumer.

Because we are defining our own programming language, we are free to introduce some translation scheme which moves the burden to implement the consumer interface for internal iterators or remembering the state for external iterators from the programmer to the compiler. For example, the C# programming language allows to implement an external iterator using a **yield return** as if it were an internal iterator [41]:

```
class EnumProducer: IEnumerable<int> {
    int n;

    EnumProducer(int n) {
        this.n = n;
    }

    public IEnumerator<int> GetEnumerator() {
        for (int i = 0; i <= n; i++) {
            yield return i;
        }
    }
}
```

Now a compiler has to generate a suitable implementation of `IEnumerator` as an external iterator which saves the state internally. A similar approach is defined for the Python programming language. While this is a convenient solution for non-recursive producers, it fails for recursive implementations.

Now for the XL programming language, we have to decide whether to use external or internal iterators. Because we have recursive structures like graphs in mind, recursive algorithms for producers will occur frequently (in particular the built-in support for queries). Compared to such algorithms, consumers will be relatively simple, for example simple loops or aggregate functions like the summation of all values. Thus it is reasonable to shift the burden from producers to consumers and to use internal iterators. To hide this from the XL programmer, the necessary consumer code is generated by a compiler for the XL programming language. For example, for the code

```
int sum = 0;
for (int value : produce(100)) {
    sum += value;
}
System.out.println(sum);
```

(which uses the syntax of the enhanced **for** statement of the Java programming language) a compiler could generate a consumer implementation like

```
final int[] sum = {0};
produce(new Consumer<Integer>() {
        public void consume(Integer __value) {
            int value = __value;
            sum[0] += value;
        }
    }, 100);
System.out.println(sum[0]);
```

I. e., the body of the loop has to be moved to the body of the `consume` method. The replacement of `sum` by an array with a single element is necessary in order to allow the consumer class to modify the value of `sum` in the enclosing block. This can be seen as a simulation of pointers like those from the C programming language. The **final** modifier is required by definition of the Java programming language, without this modifier the anonymous `Consumer` class would not be allowed to access `sum`.

The XL programming language defines several types of generator expressions. Besides generator methods like `produce` in the above example, there are also query expressions, the range operator `a : b`, the array generator `a[:]` and the guard expression `a :: b`. Expressions of type `java.lang.Iterable` can also be used as generator expressions, they yield all values of the corresponding external iterator.

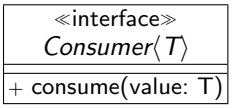
All types of generator expressions can be used in an enhanced **for** statement as shown above. They may also be combined with other expressions as in

```
produce(100) * 2
produce(100) * produce(10)
produce(produce(100))
System.out.println(produce(100));
```

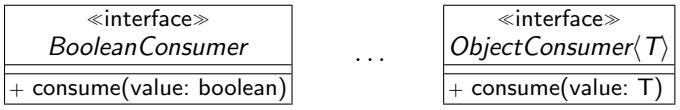
If an expression statement such as `System.out.println(produce(100));` has a generator expression, there is an implicit loop over its values.

6.3.1 Generator Methods

As generator expressions are handled by internal iterators, the declaration of *generator methods*, i. e., methods which return multiple values in succession, is easy. Such a method has to receive an additional argument for the consumer to which it can pass its return values one after another. As we have used it in the examples above, the type for a consumer could be an invocation of a generic interface declared as



But such a solution would have to box primitive values into wrapper instances which is not efficient (note that we are about to define a core feature of the XL programming language). For this reason, there is a special interface for each primitive type and a generic interface for reference types:



These interfaces are members of the package `de.grogra.xl.lang` which contains those classes and interfaces to which the specification of the XL programming language refers. Now methods with argument types

```
(BooleanConsumer consumer, ...)
...
(ObjectConsumer<? super T> consumer, ...)
```

where the ellipsis in arguments indicates an arbitrary number of further arguments are considered to be generator methods of type **boolean**, ..., `T`. If the first consumer argument is omitted in an invocation, the invocation is treated as a generator expression which yields all values to the implicit first argument for which a compiler has to provide a suitable consumer implementation as in the example on page 133.

To have a more convenient syntax for the implementation of generator methods, we introduce a syntax for a method declaration using an asterisk after the return type and the new keyword **yield** as in

```
int* produce(int n) {
    for (int i = 0; i <= n; i++) {
        yield i;
    }
}
```

which shall be equivalent to the conventional method declaration

```
void produce(IntConsumer consumer, int n) {
    for (int i = 0; i <= n; i++)
    {
        consumer.consume(i);
    }
}
```

The usage of the **yield** keyword has been adopted from the Python programming language [162]. The usage of the asterisk is suggested by its meaning "0 to n times" in regular expressions or in the notation V^* for all words (0 to n symbols) over an alphabet V.

6.3.2 Range Operator

The expression

a `:` b

using the range operator also defines a generator expression. The range operator is defined for operand types **int** and **long** and yields the values $a, a+1, \ldots, b$. If $a > b$, no value is yielded at all. Because generator expressions are allowed within an enhanced **for** statement, we may write such a loop as

```
for (int i : 0 : n) {
    ...
}
```

6.3.3 Array Generator

The array generator

a`[:]`

is a generator expression which yields all values of the elements of the array a, starting with the element at index 0. For the purpose of **for** loops, there is no need for such an expression because the Java programming language already defines the syntax **for** `(`**int** `v : a) {...}`. But it is useful for implicit loops such as `System.out.println(a[:]);` which prints all elements, or in combination with aggregate methods as in `sum(Math.abs(a[:]))`.

The expression a may also denote a generator function with **void** parameter, i. e., an instance of the interfaces `VoidToBooleanGenerator`, ..., `VoidToObjectGenerator`, see Sect. 6.13.5 on page 188. In this case, the generator methods `evaluateBoolean`, ..., `evaluateObject`, respectively, are invoked.

6.3.4 Guard Operator

The guard operator

a `::` b

defines a generator expression which yields a if b is **true** and no value at all if b is **false**. This is especially useful if a is a generator expression itself, then the guard b filters the values yielded by a. The following generator expression yields all lowercase letters from a `String` s:

```
char c;
...
(c = s.charAt(0 : s.length() - 1)) :: Character.isLowerCase(c)
```

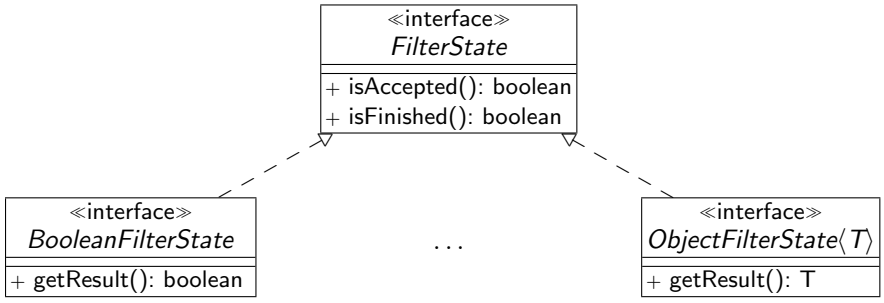
6.3.5 Filter Methods

Filter methods are a generalization of the guard operator. Their input is a sequence of values. For each value, a filter method decides whether to yield a result or not. In contrast to the guard operator, the result may differ from the input in value or even in type. For example, we may have a filter method which receives strings as input and yields their value as an **int** if they can be parsed into an **int** and nothing otherwise. Or think of a filter method `slice` such that `slice(a, begin, end)` for a generator expression `a` yields those values with positions `begin` ... `end` − 1 in the sequence.

The mechanism for filter methods invokes a filter method repeatedly with the current value of the input generator expression. From this it follows that the mechanism has to fulfil two requirements. First, there must be a possibility for the filter method to remember its state (like a counter for the `slice` method). Second, a filter method has to tell its invoker whether

- the value shall be ignored, i. e., no value shall be yielded as result of the invocation,
- a result value shall be yielded which will be the last one,
- or a result value shall be yielded and there may be further ones.

The distinction between the last two cases allows a short circuit logic. If the filter method knows that it will skip all remaining values (like the `slice` method after `end` input values), it should indicate this so that the evaluation of the input generator expression can be terminated. Both requirements are fulfilled by the introduction of the interfaces



in the package `de.grogra.xl.lang` and the convention that filter methods whose result is of primitive type look like

`F filter(F state, V value, ...)`

where F is a subtype of the corresponding interface (`BooleanFilterState`, ...), and filter methods whose result is of reference type T look like

`F filter(F state, Class<T> type, V value, ...)`

where F is a subtype of `ObjectFilterState<T>`. For both cases, `filter` stands for the method name, V is the type of the input generator expression,

and the ellipsis in an argument list indicates possible further arguments. Using this setting, the first invocation of a filter method receives **null** as `state` parameter, and all following invocations receive the return value of the previous invocation as `state` argument. Thus, a filter method can pass its state to its next invocation. For reference types, the `type` argument has to be provided so that a filter method knows the exact compile-time type of T also at run-time. Now the methods of the `FilterState` interfaces inform the invoker of the filter method about the result of filtering, whether the value has been accepted at all and if the filter is finished, i. e., if there will be no further values. A compiler has to generate code which implements this mechanism so that the programmer simply writes code like

```
slice(a, 0, n)
```

to apply a filter method

```
<T> ObjectFilterState<T> slice(ObjectFilterState<T> state, Class<T> type,
                               T value, int begin, int end)
```

to a generator expression `a`. For convenience, if the expression to be filtered is an array expression, but the filter method expects values of the type of elements of the array, then the array generator expression is implicitly used to pass the value of each element to the filter method.

The implementation of the `slice` method could look like

```
<T> ObjectFilterState<T> slice(ObjectFilterState<T> state, Class<T> type,
                               T value, final int begin, final int end) {
    class State implements ObjectFilterState<T> {
        int index = 0;
        T result;

        public T getObjectResult() {
            return result;
        }

        public boolean isAccepted() {
            return (index >= begin) && (index < end);
        }

        public boolean isFinished() {
            return index >= end;
        }
    }

    State s = (State) state;
    if (s == null) {
        s = new State();
    } else {
        s.index++;
    }
```

```
    s.result = value;
    return s;
}
```

6.3.6 Standard Filter Methods

The class `de.grogra.xl.util.Operators` contains the standard filter methods `first` and `Operators`slice for all primitive and reference types. `slice` works as the example in the previous section, while `first` is a special case thereof which filters the first n values of a sequence.

6.4 Aggregate Expressions

Generator expressions provide a convenient and efficient means to yield multiple values. Their counterpart are aggregate expressions which take multiple values in succession as input and produce a single value as result, e. g., the sum of all values. The XL programming language defines a single built-in aggregating operator (the containment operator) and a mechanism for aggregate methods.

6.4.1 Containment Operator

The containment operator

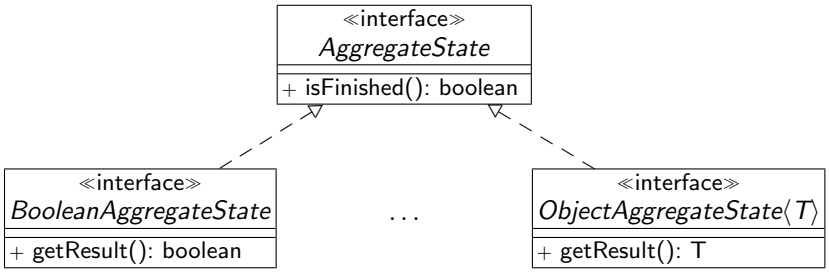
a **in** *b*

aggregates the logical or of *a* == *b*. Typical examples are `a in (1 : 3)` which has **true** as result if and only if `a` equals 1, 2 or 3, or `a in n.children()` which tests whether a node `a` is contained in the sequence of values returned by the generator method `n.children()`. The containment operator terminates the evaluation of the generator expression once it knows the final result, i. e., if *a* == *b* has been **true** for some pair *a*, *b*.

The containment operator, like **instanceof**, does not follow the general pattern that only symbols and no keywords are used for operators. However, the keyword **in** is commonly used for the containment operators, for example by the Python and D programming languages.

6.4.2 Aggregate Methods

Like filter methods, *aggregate methods* receive a sequence of values as input. However, their result is a single value. To implement the mechanism, we use a similar technique as for filter methods. Namely, we have the interfaces



in the package `de.grogra.xl.lang` and the convention that aggregate methods whose result is of primitive type look like

A `aggregate(`*A* `state, boolean finished,` *V* `value, ...)`

where *A* is a subtype of the corresponding interface (`BooleanAggregateState`, ...), and aggregate methods whose result is of reference type *T* look like

A `aggregate(`*A* `state, boolean finished, Class<`*T*`> type,` *V* `value, ...)`

where *A* is a subtype of `ObjectAggregateState<`*T*`>`. `aggregate` stands for the name of the aggregate method. Like for filter methods, the first invocation of an aggregate method receives **null** as `state` parameter, and all following invocations receive the return value of the previous invocation as `state` argument. After all input values have been passed to invocations of the aggregate method, it has to be invoked for a last time. Only for this last invocation the argument `finished` is **true** to indicate that the aggregate method should perform a final computation if any, the remaining arguments starting with `value` are set to the **null** value of their type. One could also demand that aggregate methods have to compute the current aggregated value within each invocation so that the final step is not needed, but for a complex aggregation this might be inefficient with respect to time. After the final invocation, the aggregated result is obtained by invocation of the `getResult` method of the corresponding `AggregateState` subinterface.

The method `isFinished` of the `AggregateState` interface informs the invoker of the aggregate method if the final value of the aggregated value is already known, even if there are further input values. For example, an aggregate method which returns the first value of a sequence of values already knows the aggregated value after the first input value. Now if `isFinished` returns **true**, the evaluation of the generator expression has to be terminated, and the aggregated value is obtained by the `getResult` method as before.

Again, a compiler has to generate code which implements this mechanism so that the programmer simply writes code like

```
sum(0 : 100)
```

to apply an aggregate method

```
IntAggregateState sum(IntAggregateState state, boolean finished,
                      int value)
```

to the generator expression `0 : 100`. For convenience, if the expression to be aggregated is an array expression, but the aggregate method expects values of the type of elements of the array, then the array generator expression is implicitly used to pass the value of each element to the aggregate method.

The implementation of the `sum` method could look like

```
IntAggregateState sum (IntAggregateState state, boolean finished,
                       int value) {
    class State implements IntAggregateState {
        int sum = 0;

        public int getIntResult() {
            return sum;
        }

        public boolean isFinished() {
            return false;
        }
    }

    State s = (State) state;
    if (s == null) {
        s = new State ();
    }
    if (!finished) {
        s.sum += value;
    }
    return s;
}
```

6.4.3 Standard Aggregate Methods

The class `de.grogra.xl.util.Operators` contains a collection of standard aggregate operations. Each operation is implemented by a set of methods, one for each supported source type. For a sequence of values

- the method `array` converts this into an array,
- `count` counts its number of values,
- `empty` tests if it is empty, i. e., has no values at all,
- `exist` and `forall` return the logic or or the logic and, respectively, of its boolean values (and use a short circuit logic),
- `first` and `last` return the first or last value,
- `max` and `min` the maximum or minimum value,
- `prod` and `sum` the product or the sum,
- `mean` the mean value for numeric types,
- `string` a string which contains a comma-separated list of the values, enclosed in brackets.

Furthermore, four selection operations are provided:

- `selectRandomly` selects one value out of all values based on a pseudo-random number generator. It chooses the value using a uniform distribution. A variant exists which takes a further **double** parameter which specifies the relative probability of choosing the corresponding value (e. g., `selectRandomly(x = nodes(), prob(x))` randomly chooses one of the objects yielded by `nodes`, the relative probability being given by some function `prob`).
- `selectWhere` has a further boolean parameter. The first value of the sequence for which **true** is passed to this parameter is used as the result of aggregation.
- `selectWhereMax` and `selectWhereMin` have a further numeric parameter. The value of the sequence for which the corresponding parameter value is maximal or minimal, respectively, is taken as the result of aggregation.

6.5 Queries

Queries are the most complex part of the XL programming language. A query is specified by a pattern, and its evaluation at run-time finds all matches of the pattern within the current graph. Queries can be specified as expressions on their own right (Sect. 6.5.13 on page 158), but they are also used as left-hand sides of rules (Sect. 6.8 on page 171). Examples for the first case are

```
graph.(* Node *)
(* f:F & (f.diameter > 0.01) *)
(* x:X [a:A] y:X *)
(* f:F, g:F, ((f != g) && (distance(f, g) < 1)) *)
(* x (<-successor-)* Basis *)
```

which find all nodes of type `Node` in `graph`, all nodes of type `F` with a diameter greater than 0.01, all matches of the pattern of Ex. 4.52 on page 70, all pairs `f`, `g` of distinct nodes of type `F` with a distance less than 1, or all nodes of type `Basis` which can be reached from a given node `x` by traversing an arbitrary number of successor edges backwards, respectively. The third example uses the traditional bracketed L-system syntax to textually represent a tree-like structure. An example for a complete rule is the rule for the Sierpinski grammar (Ex. 5.23 on page 113). It looks like

```
a:V -e0-> b:V -e120-> c:V -e240-> a
  ==>>
  a -e0-> ab:V -e0-> b -e120-> bc:V -e120-> c
    -e240-> ca:V -e240-> a,
  ca -e0-> bc -e240-> ab -e120-> ca;
```

where the part before the arrow symbol is a query which specifies a circular pattern.

The design of the query syntax and its semantics was the most critical part of the development of the XL programming language. On one hand, the syntax had to be intuitive and familiar to those who had worked with the L-system languages of GROGRA and cpfg. On the other hand, the syntax had to provide the possibility to specify more complex relations between elements like arbitrary edge types, relations and transitive closures thereof. And finally, the semantics should not be bound to a specific graph model, but should be defined on the basis of a rather general data model so that queries and the underlying pattern matching algorithm can not only be used for RGG graphs, which is of course the main requirement, but also for a variety of further data structures such as polygon meshes, XML documents or scene graphs of existing 3D tools. In this section, we will present and discuss the syntax for queries and their meaning in a rather informal way. A precise specification is given in [95]. In examples, we will make use of classes like `Sphere` and methods like `distance`. Although we do not define their meaning, it should be obvious what is meant.

6.5.1 Compile-Time and Run-Time Models for Graphs

At first, we have to make an abstraction for the structures on which queries of the XL programming language shall operate. This abstraction is then the common *data model* which we have to implement for concrete structures if we want to use queries for them. For the sake of readability, we will call these structures *graphs*, although they may also be just trees or even only entries in some database.

The need for a data model arises from the implementation of the built-in pattern matching algorithm of the XL programming language: there has to be a way to obtain information about the topology, namely the nodes of the current structure and, for a given node, its incident edges and adjacent nodes. To be accessible by the pattern matching algorithm, this way has to consist of universal interfaces. In general, it is common practice to define syntactic support for higher-level features via specific interfaces. E. g., the enhanced **for**-statement of the Java programming language uses the interface `java.lang.Iterable` in cooperation with `java.util.Iterator` to provide the programmer with a convenient syntax for the iteration over collections, these interfaces can be seen as a minimal data model for iterable structures.

Due to the complexity of queries, the specification of a data model has to be split into parts for compile-time and run-time. A *compile-time model* provides static information about the data model to a compiler for the XL programming language, it is represented by an implementation of the interface `de.grogra.xl.query.CompiletimeModel`. For each compile-time model, there is an associated *run-time model*, namely an instance of `de.grogra.xl.query.RuntimeModel`. Concrete graphs are instances of the interface `de.grogra.xl.query.Graph` which offers a set of requests and oper-

ations to be invoked by the run-time system of the XL programming language in order to implement the semantics of queries.

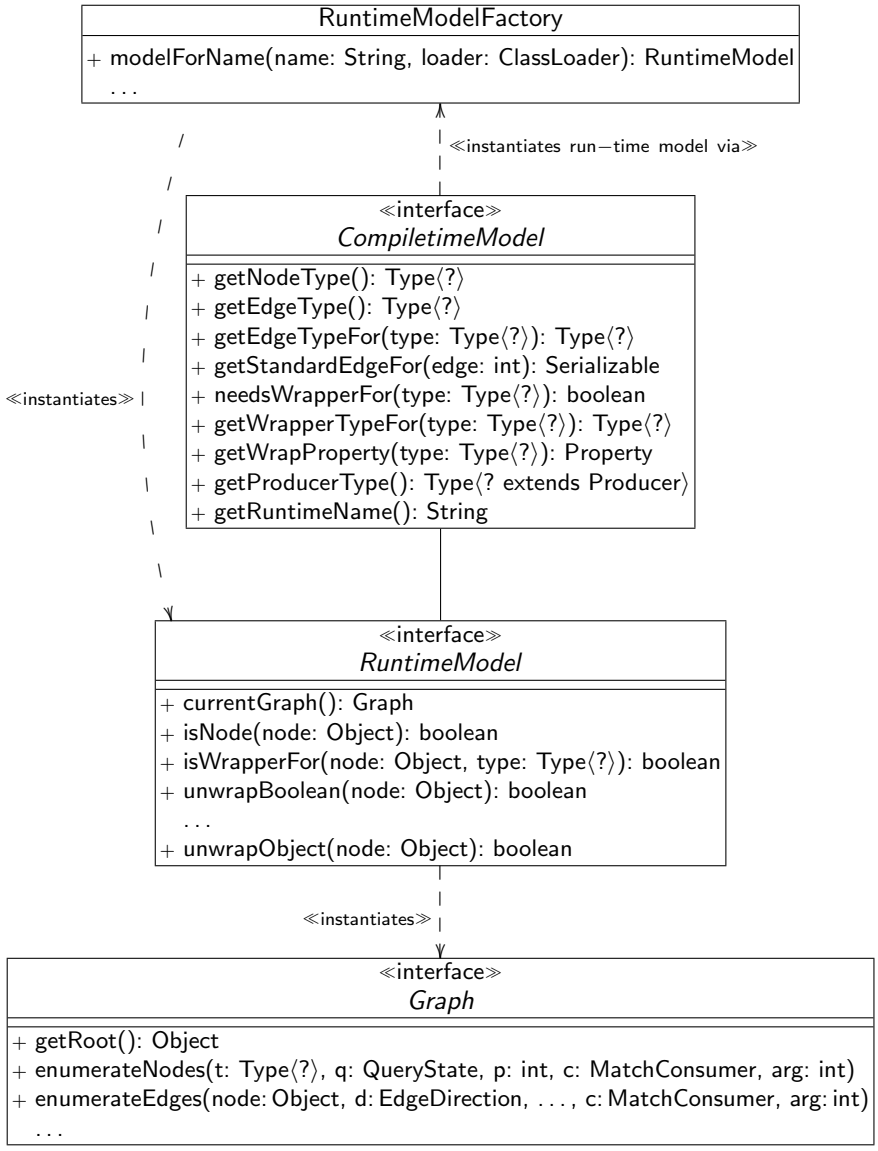


Figure 6.1. Class diagram of compile-time and run-time models for queries

Figure 6.1 on the preceding page shows the class diagram of these three interfaces. The methods `getNodeType` and `getEdgeType` of a compile-time model tell a compiler the base types of nodes and edges of the graph model, and `getRuntimeName` the name of the corresponding run-time model. The only restriction on the types is that node types must not be primitive.

When a query is evaluated at run-time, it uses a *current graph* as data source. Which graph is current is determined by the context: queries may be used as part of query expressions (Sect. 6.5.13 on page 158) or rules (Sect. 6.8 on page 171), these either specify the current graph and its compile-time model explicitly, or they may opt for an implicit determination of the current graph on the basis of its compile-time model. In the latter case, the name returned by the invocation of `getRuntimeName` on the compile-time model is used at run-time as parameter to the method `modelForName` of the class `de.grogra.xl.query.RuntimeModelFactory` in order to obtain the corresponding run-time model. On the run-time model, the method `currentGraph` is invoked to obtain the current instantiation of the model which is then taken as the current graph. An implementation of `currentGraph` could use a per-thread-association of instantiations. The method `getRoot` of a graph returns its root, the method `enumerateNodes` delivers all nodes of the graph which are instances of type `t` to the consumer `c`, `enumerateEdges` delivers all adjacent nodes of a given node where the edge matches a given condition.

We will describe more details of the models in the following sections where they are needed.

6.5.2 Node Patterns

A query pattern is composed of elementary *patterns*. The simplest patterns are patterns for nodes. If `X` is a type (in the sense of the Java programming language), then

```
X
```

within a query is a pattern which matches instances of `X`. A *query variable* is declared implicitly, the built-in pattern matching algorithm of the XL run-time system then tries to *bind* nodes of the current graph to such a variable in order to match the associated patterns. The run-time system obtains the nodes of the current graph by invoking `enumerateNodes` on it. With this simple kind of pattern, we can already specify left-hand sides of non-parametric L-system productions like `X ==> Y`. If we need to have a reference to a query variable, we can assign an identifier as in

```
x:X
```

Then `x` behaves like a final local variable, its value is the currently bound node of the structure. Note that this notation does not follow the Java programming language, which declares variables using the syntax `X x`, because this syntax is not convenient for queries: in order to be close to the notation

of L-system software, we have to use whitespace to separate node patterns (see the following section 6.5.3). But our notation is well known from, e. g., the Pascal programming language or UML diagrams.

The special node pattern

```
x:.
```

with some identifier `x` matches any node. For syntactical reasons, a stand-alone dot (i. e., without identifier) is not allowed.

Parameterized Patterns

L-system languages like those of GROGRA and cpfg also allow modules (parameterized symbols) on left-hand sides of rules as in `X(a) ==> Y(2*a)` (see Sect. 3.7 on page 25). Thus, such a pattern shall also be allowed within a query. It introduces additional query variables for each parameter, and if a node is bound to the whole pattern, the values of its attributes are bound to these parameter variables. Now there has to be a way to specify which attributes correspond to which parameters. For parametric L-systems, this is easy because modules have an ordered list of parameters whose names are simply the indices in the list. But within the context of object-oriented programming, node types have named attributes and may only want to expose some of them as parameters in a pattern, or provide several alternative patterns. So there has to be a possibility to associate parameterized patterns with a node type `X`, these patterns then define number and type of parameters and how they are linked with the attributes of `X`. For this purpose, we define a mechanism based on the abstract class `de.grogra.xl.query.UserDefinedPattern`. Each concrete subclass of `UserDefinedPattern` declares a *user-defined pattern*, where the term "user-defined" distinguishes such a pattern from built-in patterns of the XL programming language. The signature (number and types of parameters) of a user-defined pattern is given by a special method named `signature` which has no further purpose. If a user-defined pattern is declared as a member of a type `X`, then there is a pattern named `X` with the signature of the `signature` method. For the example `X(a)`, we would write code like

```
class X extends Node {
    float attr;

    static class Pattern extends UserDefinedPattern {
        private static void signature(@In @Out X node, float attr) {}

        public Matcher createMatcher(Graph graph, XBitSet bound,
            IntList requiredAsBound) {...}
        ...
    }
}
```

Then

```
X(a)
```

is a pattern whose node type is `X` and which has a parameter of type **float**. How this parameter is linked to nodes is defined by the pattern matcher which is returned by the implementation of the method `createMatcher`, this is shown in Sect. 7.4 on page 200. The `@In` and `@Out` annotations in the signature are needed when textually neighbouring patterns are connected, this is explained in Sect. 6.5.11 on page 156. It may happen that we are not interested in the values of parameters, then it is possible to write a dot instead of an identifier as in `X(.)` or even to omit parameters as in `Y(,,a,,b,)`.

Expressions as Patterns

With node types and user-defined patterns, we can specify patterns required for equivalents of L-system productions. However, in practice it may also be useful to have a pattern which only matches a fixed value. The left-hand side of the Sierpinski example `a:V -e0-> b:V -e120-> c:V -e240-> a` ends with such a pattern: the query variable `a` is used as last pattern to close the loop. But we may also think of a general expression (possibly a generator expression) whose results fix the value which has to be bound to the pattern. An example of such a general *expression pattern* is the usage of a method

```
Node* findObjectsInSphere(Tuple3d position, double radius)
```

in the pattern

```
x:findObjectsInSphere(position, radius)
```

This fixes the values of `x` to the yielded values of the invocation.

The special node pattern

```
^
```

is also an expression pattern, it matches the root node of the current graph. This is the node returned by the method `getRoot` of the `Graph` interface.

Unary Predicates

Patterns like `X` with a type `X` can be seen as predicates which test if a node is an instance of `X`. A generalization thereof is the usage of **boolean** methods as unary predicates. Say we have a method

```
boolean alive(Cell c)
```

Then we may write

```
x:alive
```

to find all nodes of type `Cell` for which `alive` returns **true**. Such a predicate may also be parameterized as in

```
x:isInSphere(position, radius)
```

for the method

```
boolean isInSphere(Node node, Tuple3d position, double radius)
```

Combining Node Patterns

Sometimes, there is a need to combine several node patterns into a single node pattern. For example, one might want to specify a pattern which finds all nodes which are instances of some class `C`, but also of some interfaces `I1`, `I2`. The Java programming language defines a syntax for the similar case where a type variable shall have a bound consisting of at most one class and several interfaces: a generic class declaration like

```
class X<T extends C & I1 & I2> {...}
```

can be instantiated for types `T` which are subtypes of `C` as well as `I1` and `I2`. We adopt this syntax and allow a `&`-separated list of node patterns as a composed node pattern. This includes simple cases like

```
x:C & I1 & I2
```

but also more complex ones like

```
x:isInSphere(position, radius) & Cylinder(l, r)
```

which finds all nodes `x` that fulfil the `isInSpere` predicate and match the `Cylinder` pattern, and which then binds length and radius of the cylinder to the query variables `l` and `r`, respectively. In addition, this list may also contain application conditions which are parenthesized **boolean** expressions: for example, the node pattern

```
f:F & (f.diameter > 0.01)
```

finds all nodes of type `F` with a diameter greater than 0.01.

Syntactical Issues

Although it is in general possible to use expressions as node patterns which fix values of query variables to their results, there are expressions which cannot be directly used for syntactic reasons. For example, if we have some array `a` and some index `i`, the expression `a[i]` cannot be used in a pattern like `X a[i] Y`: this would be syntactically interpreted as four node patterns with the third one being bracketed to define a branch (see Sect. 6.5.3 on the following page). The obvious solution to use parentheses like in `X (a[i]) Y` does not work, too: this would be interpreted as two node patterns, the first one being parameterized with parameter value `a[i]`. For such cases, we have to use a trick. The XL programming language uses backquotes to define the unary quote operator `` `a` ``. If not overloaded, it simply performs the identity operation on its operand. Expressions as node patterns may use this operator so that we can write

```
X `a[i]` Y
```

Patterns for Wrapper Nodes

Queries of the XL programming language (and the underlying data model) make a single assumption about the graph model, namely that nodes are represented as objects. However, sometimes it may be useful to store values of primitive type as nodes in the graph, or values of reference type which are not nodes themselves. If the graph model provides types for corresponding wrapper nodes, we can directly use them in the same way as normal nodes:

```
x:int
```

This pattern matches nodes which wrap **int**-values. It binds their wrapped value to `x`. Furthermore, a query variable `$x` is declared implicitly to which the wrapper node itself is bound. If `x` is used later on as a node pattern (or even on the right-hand side, see Sect. 6.7.2 on page 164), it is implicitly assumed that `$x` is meant. Whether a wrapper node is needed and which type it has is defined by the methods `needsWrapperFor` and `getWrapperTypeFor` of the compile-time model of the current graph (see Fig. 6.1 on page 143). The related method `getWrapProperty` (see again Fig. 6.1) determines the property of the wrapper type which corresponds to the wrapped value, this is discussed in Sect. 6.10.4 on page 179.

We may also use a syntax similar to parameterized predicates

```
n:int(x)      n:Comparable(x)
```

which declares query variables `n` for the wrapper node and `x` for the wrapped value. If the type of wrapped values could also be a node type, this variant is necessary to force the wrapping mechanism. For example, the type `Comparable` of the second example is an interface. In a pattern `x:Comparable` this could be interpreted as a pattern for nodes implementing `Comparable` and not as a pattern for nodes wrapping a `Comparable`.

6.5.3 Path Patterns

While 0L-systems only allow a single symbol on their left-hand side, we allow arbitrary graphs for left-hand sides of rules of relational growth grammars. In order to specify these graphs, we have to connect node patterns with edge patterns or, more general, with path patterns which may encode arbitrary relations (see also the discussion about relations in Sect. 5.7 on page 114). For a query in the XL programming language, the simplest path pattern is to not specify a path pattern at all and simply have two node patterns with only whitespace between as in `X Y`. On the right-hand side of an L-system production, this would stand for two successive symbols, their graph representation according to Def. 5.2 on page 96 would consist of two nodes

connected by a successor edge. So we adopt this notation: if two node patterns are separated by whitespace only, there is an implicit edge pattern between them which matches a successor edge from the node bound to the left pattern to the node bound to the right pattern. We also adopt the bracket notation: a node pattern followed by a bracketed pattern starting with a node pattern implies an edge pattern which matches a branch edge from the node bound to the first node pattern to the node bound to the bracketed node pattern. In general, a pattern which is a well-nested word over the alphabet of node patterns implies those edge patterns which correspond to edges of its graph representation.

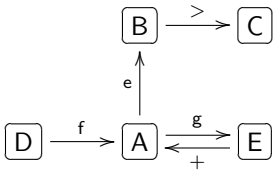
Explicit Path Patterns

In order to be able to specify relations explicitly, path patterns of the form

`-`*r*`->` `<-`*r*`-` `-`*r*`-` `<-`*r*`->`

are allowed where *r* specifies which relation to use. The first two variants are used for directed relations, the third one for undirected relations, the last one for bidirectional relations. This intuitive syntax follows the PROGRES language (Sect. 4.8.1 on page 82). Like for node patterns, *r* may refer to a type of the Java programming language (the type of matching edges), it may denote a user-defined pattern, or it may be some expression. The interpretation of the latter case depends on the implementation of the method `enumerateEdges` of the current graph: the result of an expression *r* may be interpreted either as an edge object, in which case it has to be checked whether its incident nodes match the nodes bound to incident node patterns, or it may be interpreted as a pattern itself which selects a subset of the edges of nodes bound to incident node patterns. For example, within the GroIMP software (Appendix B) the last option is used: edge types are represented by **int** values, and we have, e. g., the predefined constant `successor` which stands for successor edges. Thus, a path pattern for successor edges would be specified as `-successor->`, i. e., the value of `successor` is used as a pattern which matches only specific **int** values as edge types. The pattern `-successor-` matches if there is a successor edge in one or the other direction, while `<-successor->` requires edges in both directions.

If a bracketed pattern starts with a path pattern, the latter overrides the default behaviour of assuming a branch edge. As an example, the pattern `a:A [-e-> B C] [<-f- D] -g-> E [a]` corresponds to a pattern graph



Standard Edge Patterns

There are the special shorthand notations

```
-->     <--     --      <-->
```

which stand for edge patterns for any edge in the indicated direction, and the notations

```
>       <       ---     <->
+>      <+      -+-     <+>
/>      </      -/-     </>
```

which stand for edge patterns for successor, branch or refinement edges, respectively, in the indicated direction.

The precise meaning of these edge types is not defined by the XL programming language. This is rather left to the current compile-time model: for the corresponding constants `ANY_EDGE`, `SUCCESSOR_EDGE`, `BRANCH_EDGE`, `REFINEMENT_EDGE` the invocation of `getStandardEdgeFor` on the model returns a representation of the edge type in the used graph model, this representation is taken as the pattern expression for the path pattern.

Binary Predicates

As for node patterns, the possibility to use **boolean** methods as predicates is convenient (see also Def. 5.26 on page 114). Therefore, if we have a method like

```
boolean neighbour(Node a, Node b)
```

we may simply write

```
X -neighbour-> Y
```

as path pattern. Note that we have the same syntax for true edges and binary predicates (relations), although (within the framework of relational growth grammars) the first case is handled by edges of the graph of the left-hand side, while binary predicates have to be represented as application conditions.

Node Functions as Path Patterns

Relations may also be expressed by functions which map a node to a set of related nodes, see Def. 5.27 on page 115. Such a function is implemented as a method like `Node* neighbours(Node a)` or, in case of at most one possible related node, `Node parent(Node a)`, and we can specify a path pattern `X -neighbour-> Y` using the same syntax as for binary predicate methods. For both cases, we may also have additional parameters as in

```
X -neighbour(1)-> Y
```

for a method

```
Node* neighbours(Node a, double radius)
```

Furthermore, the method may have an additional last parameter of type `de.grogra.reflect.Type`. Then an implicit argument is provided which represents the type of the query variable to which the result of the method invocation is bound. For example, a method `Node* neighbours(Node a, Type t)` could look for all neighbours of type `t`, and its usage in `X -neighbour-> Y` would provide the type `Y` to the method.

6.5.4 Composing Patterns

Node patterns, combined with path patterns, define a connected pattern graph. Path patterns may be concatenated without node patterns between as in

```
A > > +> > B
```

which finds all nodes of types `A`, `B` that are connected with a path of two successor, a branch and a further successor edge. So far, there is no possibility to define unconnected pattern graphs simply because if there is no explicit path pattern between two node patterns, an implicit pattern for successor edges is inserted. In order to compose several unconnected parts to build a complete pattern graph, a comma is used as a separator like in the example

```
x:X, y:Y
```

which finds all pairs of nodes of types `X` and `Y`, respectively, and does not impose any relation between the matches `x` and `y`. If a part of such a compound pattern starts with an opening parenthesis, this part is considered to be an application condition, i. e., a **boolean**-valued expression. An example is

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

which finds all pairs `f`, `g` of distinct nodes of type `F` with a distance less than 1. An equivalent pattern could be specified by moving the condition to the second node pattern as in `f:F, g:F & ((f != g) && (distance(f, g) < 1))`, but this variant hides the symmetry of the condition.

6.5.5 Declaration of Query Variables

A query may start with the declaration of query variables in the usual syntax of the Java programming language. The sole purpose of this is to shorten the specification of a query with several variables of the same type. For example, the left-hand side of the crossing-over production on page 103 could be specified as

```
i:Gene j:Gene, k:Gene l:Gene, i -align- k
```

if `Gene` is the class for gene nodes. If we declare the query variables in advance, the code becomes shorter:

```
Gene i, j, k, l;
i j, k l, i -align- k
```

Because values are bound to query variables by the built-in pattern matching algorithm, it is not allowed to specify initial values in initializers such as `Gene i = null;`.

6.5.6 Transitive Closures

Transitive closures of relations (see Def. 5.28 on page 115) are often needed, so their inclusion in the query syntax is desirable. For example, within a plant model we might want to know all descendants of some given node `p` which are of type `Internode`. Being a descendant is determined by the transitive closure of being a child. Assuming a suitable relation `child`, we write

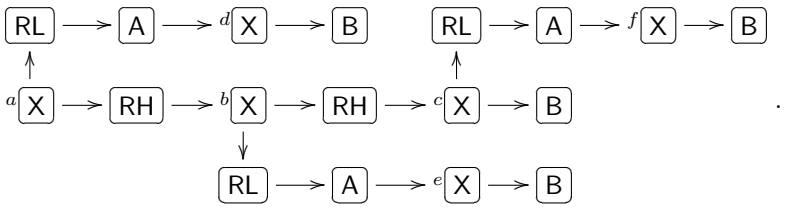
```
p (-child->)+ Internode
```

Here, we adopt the well-known syntax of regular expressions where patterns to be repeated are enclosed in parentheses with a quantifier appended. Possible quantifiers are

```
+      *      ?      {n}      {n, m}      {n,}
```

The quantifier `+` stands for 1-to-n repetitions, this yields the transitive closure. The quantifier `*` stands for 0-to-n repetitions, thus it corresponds to the reflexive, transitive closure. The remaining quantifiers only specify subsets of transitive closures: the quantifier `?` denotes 0-to-1 repetitions, the quantifier `{n}` forces exactly `n` repetitions, `{n, m}` stands for `n`-to-`m` repetitions, and `{n,}` for at least `n` repetitions. The contained relation of such a closure may be arbitrarily complex: e. g., `(-branch-> b:B (-successor->)*, (b.prod > 0))*` matches all paths which traverse a `branch` edge and a possibly empty sequence of successor edges alternatingly. In addition, target nodes of `branch` edges have to be of type `B` and fulfil the condition `b.prod > 0`.

It is often the case that one is not interested in the complete transitive closure, but only wants to find the nearest occurrences of some pattern when traversing paths of a given basic relation. For example, assume that we want to implement transport or signalling in virtual plants with a tree-like structure. These processes shall use nodes of type `X`, but the plants do not only consist of such nodes, but also of several nodes of other types between the `X`-nodes like in the tree



Then for a local transport, only the nearest `X`-neighbours have to be considered. Thus, for a basipetal (downwards) transport, node c should consider node b, but not node a, and for an acropetal (upwards) transport, node a should consider nodes b and d, but not the other ones. Using the edge pattern `-->` to match the edges of the tree structure, the transitive closures `(<--)+` and `(-->)+` would find these neighbours, but then traverse even deeper into the structure. However, we are only interested in *minimal elements*. This can be specified with the help of the syntax

```
(<--)+ : (X)      (-->)+ : (X)
```

which tells the transitive closure to stop deepening the closure once a match of the parenthesized pattern after the colon has been found. This can be compared to non-greedy pattern repetitions for minimal matching within regular expressions of the Perl programming language [145].

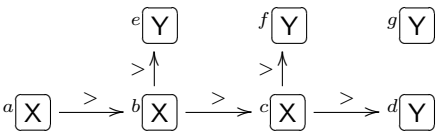
Usually, the pattern matching algorithm enforces the constraint of injectivity of a match with respect to nodes (Sect. 7.3.4 on page 199). This would exclude reflexivity of (subsets of) transitive closures, as reflexivity means that both related node variables may be bound to the same node. Thus, practically the meanings of the quantifiers `*` and `+` would coincide. To circumvent this unwanted behaviour, an implicit folding clause (see Sect. 6.5.9 on page 155) is added for the two node patterns incident with a transitive closure having a quantifier different from `+`.

6.5.7 Single Match, Late Match and Optional Patterns

There are usually several matches for patterns or components of patterns. If we want to restrict the set of matches to the first match, we write a *single match* pattern

`(:` *pattern*`)`

where *pattern* is some pattern. For example, the pattern `X (: Y)` finds nodes of type `X` from which we can reach a node of type `Y` by traversing a successor edge. Even if there are several `Y`-children of an `X`-node, at most one total match per `X`-node is yielded. For the graph



the pattern `X Y` yields the matches $(b, e), (c, d), (c, f)$, while `X (: Y)` yields $(b, e), (c, d)$ or $(b, e), (c, f)$, depending on whether d or f is found at first. The latter depends on the internal implementation of the pattern matching algorithm and an order of edges within the graph and is not specified by the XL programming language. There is some further speciality of the single match

pattern: so far, all presented patterns could be combined to a compound pattern, and the order in which the pattern matching algorithm tried to match the individual patterns did not influence the set of total matches of the compound pattern (of course with the exception of possible side effects due to the invocation of methods). But as one can see from the example, matching of single match patterns has to be done as late as possible in order to be useful. Otherwise, we could start with binding g to `Y`. Then we would not find an `X`-partner and actually no match at all since `Y` is enclosed in a single match pattern. Matching as late as possible means that if a compound pattern contains some single match pattern p, matching of p does not start unless all other non-single-match components which do not depend on the match of p have been matched.

The possibility to control the order of matching may be useful in general. Some necessary constraints on the order might follow from side effects which the pattern matching algorithm does not know. Or without a hint, it may happen that the algorithm chooses a disadvantageous order with respect to computation time. For such purposes, we use the *late match* pattern

`(&` *pattern*`)`

which is matched as late as possible just like the single match pattern, but has no restriction concerning the number of matches.

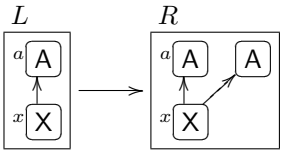
A very useful variant of the *late match* pattern is the *optional pattern*. It is also matched as late as possible, but if no match for the contained pattern could be found, the optional pattern matches nevertheless. The syntax is

`(?` *pattern*`)`

The pattern `X (? Y)` would yield the matches $(a, \textbf{null}), (b, e), (c, d), (c, f)$ for the graph from above. As we can see, the value **null** is bound to query variables of non-matched optional patterns if the variables have a reference type. Likewise, **false**, 0 and Not-a-Number are used for **boolean**, integral and floating-point types, respectively. The optional pattern may be combined with the first match pattern as in `X (:? Y)` or the equivalent variant `X (?: Y)` so that if there are matches, only the first one is used.

6.5.8 Marking Context

The notion of *context* in the rule-based paradigm refers to some structure which has to exist within the current host structure and which is related in some way with the match, but which is not replaced on rule application. For example, for the SPO production



the left-hand side L is also the context graph since all objects of L reappear on the right-hand side. In a way the node a and its incoming edge are even "more context" than the node x since x receives a new outgoing edge on derivation. A textual notation of the rule (see Sect. 6.8 on page 171 for the syntax of the right-hand side) like

```
a:A < x:X ==>> a < x A;
```

is the direct representation of the production, but lengthy and error-prone (we have to exactly repeat parts of the left-hand side). Therefore, parts of a query may be marked as context by enclosing them in asterisked parentheses: both rules

```
(* A < *) x:X ==>> x A;
(* A < x:X *) ==>> x A;
```

are equivalent to the original production, given that the underlying implementation of the rewriting mechanism has a suitable behaviour.

6.5.9 Folding of Query Variables

The pattern matching algorithm obeys several constraints. One constraint is the injectivity of matches with respect to nodes, i. e., no two query variables containing nodes may be bound to the same node of the graph. For the details, see Sect. 7.3.4 on page 199. However, there are situations where this constraint it too restrictive. One solution is then to completely disable the requirement of injectivity, but a more fine-grained mechanism is given by *folding clauses*, a feature which is also available in the PROGRES software [175]. Folding clauses are suffixes of whole node patterns, separated by the symbol `|`, and list the identifiers of those preceding node patterns which may be bound to the same node:

```
a:Node --> b:Node --> c:Node|a
a:Node, b:Node, c:Node|a|b
```

The first pattern looks for a sequence of connected nodes which may be a cycle (`c` may coincide with `a`). The second pattern searches for three unrelated nodes where `c` may coincide with both `a` and `b`. Note that this also means that `a` may coincide with `b` in the case that `c` coincides with `a` and `b`.

6.5.10 Query Initialization

If a whole query starts with an opening parenthesis, the parenthesized part has to be a comma-separated list of expressions. These are evaluated once at the beginning of the execution of the query and may serve to initialize the query. Namely, within the parentheses the current query state (Sect. 7.1 on page 193) is in scope in the same way as in with-instance expression lists (Sect. 6.13.4 on page 187). Now methods which the query state provides to

configure its behaviour can easily be invoked. For example, a query state may provide a method `noninjective` which disables the default constraint of injective matches, then a query

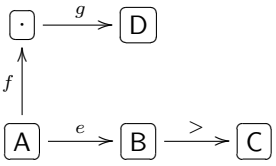
```
(* (noninjective()) a:A, b:A *)
```

also finds matches where `a` = `b`.

6.5.11 How Patterns are Combined

The combination of node patterns and path patterns to a compound pattern graph has been informally described in the previous sections. We will give some more details in this section, a thorough discussion is given in [95].

The query syntax is used to define pattern graphs, but as it is textual, these graphs have to be given in a serialized way. A very basic way of such a serialization would be to list the components, and to explicitly specify their common parts by identifiers for the gluing points. For example, to specify a pattern graph



where the dot stands for an arbitrary node type, we could use a syntax like

```
a:A, b:B, c:C, x:., d:D, a -e-> b, b -successor-> c, a -f-> x, x -g-> d
```

This is inconvenient in practice. In fact, we have not exploited the possibilities of a textual notation: in a text, each symbol may have a left and a right neighbour. This neighbourhood can be used to represent connections in an intuitive, textual way as it has been presented in the previous sections. For the example, two possible textual notations are

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

This syntax requires three underlying conventions:

- When a path pattern like `-e->` is neighbouring with a node pattern, it shares the query variable of the node pattern. This glues both patterns.
- When two path patterns are neighbouring, an implicit node pattern which matches any node is inserted between.
- When two node patterns are neighbouring, an implicit path pattern for successor or branch edges is inserted, depending on whether there is an opening bracket between the patterns.

Here, we also have to consider patterns as neighbouring if they are only separated by complete bracketed patterns and, optionally, a final opening bracket.

These conventions are implemented by the following definitions: A *pattern* (be it a node pattern or a path pattern) has a number of *parameters* (at least one) which are linked with the query variables of the query. For example, a type pattern `X` has a single parameter of type `X`, a parameterized pattern `X(a)` has two parameters (one for the node and one for the parameter `a`), an edge pattern `-->` has two parameters for its incident nodes. A pattern either distinguishes a single parameter as its *in-out-parameter*, then it is a node pattern, or it distinguishes a parameter as its *in-parameter* and a different one as its *out-parameter*, then it is a path pattern. If a pattern p is followed by a pattern q and at least one of p, q is a path pattern, the out-parameter of p is glued with the in-parameter of q, i. e., both share the same query variable. Otherwise, both patterns are node patterns, and they are joined by an implicit edge pattern for successor or (if there is an opening bracket) branch edges whose in-parameter is glued with the out-parameter of p and whose out-parameter is glued with the in-parameter of q. For built-in patterns of the XL programming language (i. e., all presented patterns with the exception of user-defined ones), which parameter is the in-parameter and which is the out-parameter is quite obvious and defined in the specification [95]. For user-defined patterns (see Sect. 6.5.2 on page 145), this is defined by annotating the corresponding parameters in the `signature` method with the annotations `@In` and `@Out` as in

```
class ParameterizedX extends UserDefinedPattern {
    private static void signature(@In @Out X node, float attr) {}
    ...
}

class Neighbours extends UserDefinedPattern {
    private static void signature(@In Node a, @Out Node b,
                                  double distance) {}
    ...
}
```

6.5.12 Declarations of User-Defined Patterns

The declaration of user-defined patterns as subclasses of `UserDefinedPattern` (see Sect. 6.5.2 on page 145, Sect. 7.4 on page 200 and the previous section) is very flexible, but there is no syntactical support for an easy implementation of such patterns (we have to use imperative code for this). For the purpose of reusability, a simple syntax is desirable which allows one to specify user-defined patterns composed of other patterns. These could then be reused as part of other patterns just like a method can be invoked from several places. The syntax to do this is shown by the following example:

```
class xyzPath(@In Node a, @Out Node b) (
    a -x-> -y-> -z-> b
```

```
)

... X -xyzPath-> Y ...
```

A subclass `xyzPath` of `de.grogra.xl.query.UserDefinedCompoundPattern` (which is a subclass of `UserDefinedPattern`) is declared. Its `signature` method has the signature `(@In Node a, @Out Node b)`, and it represents a compound pattern `a -x-> -y-> -z-> b`, i. e., a path from `a` to `b` along three edges of types `x`, `y`, `z`, respectively. The last line of the example uses this pattern.

6.5.13 Query Expressions

A query can be used as an expression if it is enclosed in asterisked parentheses like

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

Such an expression finds all matches of the pattern by the algorithm described in Chap. 7 and yields the currently bound value of the right-most non-bracketed node pattern. It is a generator expression by its very nature. The current graph for such a query expression is defined implicitly: there has to be an enclosing declaration which has an annotation of type `@de.grogra.xl.query.UseModel`. This annotation has a single element which has to specify a concrete class to use as compile-time model. As an example, the code

```
@UseModel(MyCompiletimeModel.class)
class Simulation {
    void check() {
        double len = sum((* F *).length);
        ...
    }
}
```

would use an instance of `MyCompiletimeModel` as compile-time model for the query. This then defines a current graph to be used at run-time, see the discussion of the mechanism in Sect. 6.5.1 on page 142.

If we want to specify the current graph explicitly, it has to be prepended in front of the query as in

```
graph.(* ^ (>)* Node *)
```

Then the current graph is the result of the prepended expression. The type of the expression has to be a subtype of `de.grogra.xl.query.Graph`, and it has to have or inherit an annotation of type `@de.grogra.xl.query.HasModel` whose single element specifies the corresponding compile-time model. An example would be

```
@HasModel(MyCompiletimeModel.class)
class MyGraph implements Graph {...}

class Simulation {
    void check(MyGraph graph) {
        double len = sum(graph.(* F *).length);
        ...
    }
}
```

Of course, specifying the current graph explicitly is principally preferable compared to having a current graph on some global basis. But since typical applications like plant models work with a single graph throughout their lifetime, having an implicit graph or not is a matter of convenience.

Query expressions in combination with aggregate methods are a flexible tool to specify globally sensitive functions. This has also been discussed from an abstract point of view in Sect. 5.1 on page 89, but now we can verify this from a practical point of view. For example, function 2 of the GROGRA software computes the minimal distance from a given elementary unit `c` (the turtle interpretation of an F symbol) to all other elementary units `x`, excluding the parent unit and units shorter than a threshold `t`. Using a query expression and an aggregate method `min`, this can be specified by

```
min(distance(c, (* x:Shoot, ((x != c) && (x.getLength() >= t)
                             && (x != c.getParentShoot())) *)))
```

so that such a function is no longer a fixed part of the used modelling software, but a part of the model itself.

Arithmetical-structural operators (Sect. 3.11 on page 30) were defined for the GROGRA software to allow some amount of model-specific global sensitivity, where the globality is restricted to all descendants of the current elementary unit, all ancestors, or direct children, depending on the used operator. Within the following piece of GROGRA source code

```
\var len length,
\var dia diameter,
...
sum(dia > 0.01, len)
```

the last expression uses the **sum** operator to compute the sum of the lengths of all descendants of the current elementary unit `c` whose diameter exceeds 0.01. Such a computation can also be expressed using queries and aggregate methods:

```
sum((* c (-->)* s:Shoot & (s.getDiameter() > 0.01) *).length)
```

6.6 Operator Overloading

A lot of modern programming languages support *operator overloading*, i. e., the possibility to assign new domain-specific meanings to operators. In the C++ programming language [83], we may implement a class for complex numbers like

```
class Complex {
    public:
        double real;
        double imag;

        Complex operator+(Complex b) {
            return Complex(real + b.real, imag + b.imag);
        }
    ...
}
```

and then simply write `a + b` for two `Complex` numbers `a`, `b` to compute their sum according to the implemented overloading of the operator `+`.

Unfortunately, for the sake of simplicity the Java programming language does not provide the programmer with a possibility to define overloading. This does not restrict the expressiveness of the language because we may invoke normal methods like `add` to perform specific operations, but it surely restricts the conciseness. Were it only for this reason, the definition of operator overloading for the XL programming language would be desirable, but not mandatory. However, using operator overloading we can define the meaning of right-hand sides of rules in a very general and versatile way. How this is done precisely is given in the next sections, but a short example helps to get an idea. Say that we want to have a syntax for right-hand sides similar to the syntax for left-hand sides. Thus, typical right-hand sides of L-system productions like `RH(137) [RU(60) M] F(1) M` shall be allowed and have the corresponding meaning, but also right-hand sides with an extended syntax like

```
x [-e-> Y] < Z, x W
```

The implementation of the actual actions to be taken for a right-hand side is most suitably handled by some "producing device" `producer`, this could look like

```
producer.addNode(x).push().addEdge(e).addNode(new Y()).pop()
    .addReverseEdge(successor).addNode(new Z()).separate()
    .addNode(x).addNode(new W())
```

with the assumption that the used methods return the `producer` itself or some equivalent object so that we can do "invocation chaining". While this is already quite general because the actual operations are defined by the concrete method implementations of the `producer`, a more elegant solution is to regard

right-hand sides as consisting of operators and expressions with a special syntax such that symbols like `[` or `<` are simply operators for which a suitable overload has to exist. Using the translation scheme defined in the next section, the translation of the example results in

```
tmp1 = producer.producer$begin().operator$space(x);
tmp2 = tmp1.producer$push().producer$begin().operator$arrow(new Y(), e);
tmp2.producer$end();
tmp1.producer$pop(tmp2).operator$lt(new Z()).producer$separate()
    .operator$space(x).operator$space(new W()).producer$end()
```

As can be seen from the example, operator overloading in the XL programming language works with the help of *operator methods* whose name is composed of the prefix `operator$` and a suffix for the operator (like `lt` for `<` or `add` for binary `+`). These methods can be declared either by a usual method declaration like

```
Complex operator$add(Complex b) {...}
```

or, equivalently but more conveniently, by an operator method declaration

```
Complex operator+(Complex b) {...}
```

Although the `$` character is a legal part of an identifier, there is a convention that it should only be used within compiler-generated names [72]. This is the case for operator method declarations, so the usage of `$` within names of operator methods is justified and prevents a name collision with non-operator methods.

Now if there is an expression like `a + b` for which the XL programming language does not define a meaning by itself, it has to be checked whether applicable operator methods exist. For the example `a + b`, this could be an instance method `operator$add` of the compile-time type of `a` with a single parameter whose type is compatible with `b` so that the expression becomes translated to `a.operator$add(b)`. Or there could be a method named `operator$add` within the current scope which has two parameters of suitable type, then the expression becomes translated to `operator$add(a, b)`. Finally, also static methods declared in or inherited by the types of a and b are considered. Unary operators are treated in a similar way. The operator `[]` is extended from a binary operator `a[i]` to an operator of arbitrary arity (at least binary) `a[`$\mathtt{i}_1$`, ..., `$\mathtt{i}_n$`]`.

Method invocations in the Java programming language look like $q.n(a)$ or $n(a)$, where q is an expression or a type, n a simple name of a method and a a list of arguments. For convenience, also such an invocation operator `()` shall be overloadable, but this is different from the previously described cases: for a method invocation the operands $q.n$ or n are no valid standalone expressions since the Java programming language has no built-in notion of method-valued expressions (unlike, e. g., the C++ programming language where n would evaluate to a pointer to function n). So only if there is no

valid method invocation $q.n(a)$ or $n(a)$, it is tried to interpret $q.n$ or n as an expression and to find a suitable operator method `operator$invoke`.

A special handling is required for the operators `==`, `!=` and **in**. Since these are defined for almost every type (except pairs of operand types which cannot be tested for equality, for example **boolean** and **int** or `String` and `Number`), the precedence of built-in operator semantics over operator methods would preclude the usage of corresponding operator methods. While one may argue that the overloading of `==` and `!=` is no good idea, this definitively does not hold for the **in** operator which could, e. g., be overloaded to test the containment of an element in a set. For this reason, operator methods for these operators have precedence over the built-in semantics.

In the Java programming language, an expression statement like

```
1 << 2;
```

would result in a syntax error. Although the operation is valid, its usage as a statement is useless as the result is discarded and the operation has no side effects. To detect such apparent programming errors, the designers of the Java programming language chose to forbid this and similar constructs already at syntactical level. However, with the possibility of operator overloading a statement like

```
out << 2;
```

makes perfectly sense so that the XL programming language allows any expression to be used in an expression statement.

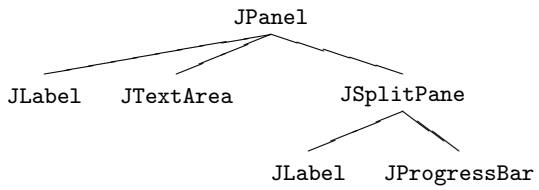
6.7 Production Statements

While left-hand sides of rules are specified by queries, right-hand sides consist of *production statements* which have a special syntax and are mostly defined by operator overloading. As an example for a complete rule, the representation of the movement production on page 103 within the XL programming language is given by the following code if we use successor edges for the chain of `X`-nodes and branch edges to indicate where `A`-nodes are located:

```
x:X [a:A] y:X ==>> x y [a];
```

The execution of such a rule finds all matches of the left-hand side of the rule arrow `==>>`, and for each match the production statements of the right-hand side (interpreted using operator overloading) are executed. But similar to queries which may also be used outside of rules, production statements do not only occur within replacement rules, but also as part of instantiation rules of module declarations (Sect. 6.11.2 on page 182), and as part of normal imperative code (Sect. 6.9 on page 174). The latter is useful if we want to create a graph-like structure outside of a rule. In order to have not only modelling-related examples, consider components of a GUI framework. These are typically composed to a tree as in the Swing example

```
JPanel panel = new JPanel(new BorderLayout());
JLabel lbl = new JLabel("Message");
JTextArea text = new JTextArea();
JLabel status = new JLabel("Type in a message.");
JProgressBar progress = new JProgressBar(0, 100);
JSplitPane split = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
split.addComponent(status);
split.addComponent(progress);
panel.addComponent(lbl, NORTH);
panel.addComponent(text);
panel.addComponent(split, SOUTH);
```



Using production statements within imperative code, this could be written in a more convenient way:

```
awtProducer ==> panel:JPanel(new BorderLayout())
    [-NORTH-> JLabel("Message")]
    [JTextArea]
    [-SOUTH-> JSplitPane(JSplitPane.HORIZONTAL_SPLIT)
                [JLabel("Type in a message.")]
                [JProgressBar(0, 100)]];
```

where `awtProducer` implements the mapping from production statements like `-NORTH-> JLabel("Message")` to corresponding method invocations (in this case `panel.addComponent(lbl, NORTH)`, see next sections). Note that this example is not a rule, but a normal statement which can be specified as part of a usual code block.

The actual effects of the execution of production statements are not defined by the XL programming language, they depend on the implementation of special operator methods. For example, these methods may immediately create nodes and edges in a graph. This is useful for cases like the previous where a graph has to be constructed outside of a rule. Within rules such an immediate mode could be used for sequential derivations, while parallel derivations require that the actions are collected and applied at a later point in time.

The most important production statements are node expressions like `panel:JPanel(new BorderLayout())`, possibly prefixed by path expressions like in `-NORTH-> JLabel("Message")`. The above examples also show the possibility to specify branches in brackets. In addition to these types of production statements, we may embed conventional code blocks, and we may use control flow statements. Within rules, the latter provide a means for the dynamic

creation of the successor as described in Sect. 5.4 on page 105. The precise mechanism of production statements is described in the next sections.

6.7.1 Execution of Production Statements and Current Producer

Contrary to queries, production statements do not make any reference to run-time models. But within the context of production statements, a *current producer* is defined which serves a similar purpose. It is an object according to the Java programming language, the type of which is not restricted by the specification of the XL programming language. The initial current producer is given by the rule or the special production block which contains the production statements. On execution, each production statement takes the current producer as input, invokes some method on the producer, and provides a new current producer for subsequent production statements. For an example, reconsider page 161.

At the beginning of the outermost sequence of production statements (i. e., at the beginning of a right-hand side, of production statements within imperative code, or of an instantiation rule), the method `producer$begin` is invoked on the current producer. It is a compile-time error if this method does not exist. The result of the invocation is taken as new current producer. At the end of the outermost sequence of production statements, the method `producer$end` is invoked on the current producer. Again, it is a compile-time error if this method does not exist.

Within production statements, fields and methods declared or inherited by the type of the current producer can be referred to by their simple name. For instance fields and methods, the current producer is used as the implicit target reference similar to the implicit **this** in bodies of instance methods. However, these fields or methods do not shadow members with the same name in enclosing scopes, i. e., the latter have higher priority.

6.7.2 Node Expressions

Node expressions are used to specify nodes within production statements. The introductory example for a rule

```
x:X [a:A] y:X ==>> x y [a];
```

contains three simple node expression on its right-hand side which refer to the value of query variables, i. e., to matches of the left-hand side. The node expressions are separated by whitespace and brackets to create successor and branch edges, how this works in detail is described in the next section.

More general, we may use arbitrary non-**void** expressions as node expressions like in the rules

```
x:X [a:A] ==>> x, Model.findNeighbours(x).chooseOneFor(a) [a];
x:X ==>> x new Y(42);
```

Rules often create new nodes. For example, L-system productions, translated to graphs, do only specify new nodes. The L-system production for the Koch generator

$$\mathsf{F} \rightarrow \mathsf{F}\ \mathsf{Left}\ \mathsf{F}\ \mathsf{Right}\ \mathsf{Right}\ \mathsf{F}\ \mathsf{Left}\ \mathsf{F}$$

removes a node of type F and creates four new nodes of type F, two new nodes of type Left and two new nodes of type Right. We could write such a right-hand side as

```
new F() new Left() new F() new Right()
new Right() new F() new Left() new F()
```

However, this is very inconvenient in practice. Therefore, we introduce the convention that the keyword **new** and empty pairs of parentheses can be omitted. Then, the right-hand side of the Koch rule can be written as a sequence of production statements

```
F Left F Right Right F Left F
```

The downside is that this introduces some amount of ambiguity since `F` may denote both a type of which the constructor has to be invoked, a method or a variable, but accepting this drawback is justified from a practical point of view. How ambiguities are resolved is defined in the specification of the XL programming language [95].

As for node patterns, a node expression may be prefixed by an identifier and a colon. This declares a new local variable whose name is given by the identifier and whose value is the result of the node expression. This can be used if we need to reference a node at another code location, for example if we want to create a loop within production statements:

```
a:A A A A a
```

Again as for node patterns, the `^` character may be used as a node expression which returns the root of the current graph, or, to be more precise, the result of the invocation of the method `producer$getRoot` on the current producer. This is useful within rules if we want to attach new nodes to the root, irrespective of the location of the match of the left-hand side in the graph. For example, in a plant model a fruit may become a seed which detaches from the plant and gives rise to independent offspring. This could be specified as a rule

```
Fruit ==>> ^ Seed;
```

Like for node patterns, some expressions cannot be used directly for syntactical reasons. We have to use the same trick, namely applying the quote operator `` `a` `` to the expression.

Within production statements, there is also support for wrapper nodes. First, if the left-hand side of a rule specifies a wrapper node pattern like `i:`**`int`** and we use `i` as a node expression within the right-hand side, the wrapper node `$i` is used instead (see Sect. 6.5.2 on page 148 for wrapper nodes in queries). So if **int** is used to represent gene values, the crossing-over production on page 103 could be specified as

```
int i, j, k, l;
i j, k l, i -align- k ==>> i l, k j;
```

If an expression which would normally be treated as a node expression has **void** as its type, this is handled differently for convenience: such an expression is simply evaluated and has no further influence on the execution of the production statements. A typical usage is for messages as in the rule

```
s:Sphere ==> b:Box println("Replaced " + s + " by " + b);
```

6.7.3 Prefix Operators for Node Expressions

Each node expression is prefixed by an operator. Both together constitute a single production statement which passes the result of the node expression to the current producer. In the simplest case, node expressions are prefixed by whitespace only as in the Koch rule

```
F ==> F Left F Right Right F Left F;
```

For such a node expression, a method `operator$space` is invoked on the current producer p with the result of the node expression as argument. The result of the invocation is taken as new current producer p'. For the example, this effectively leads to a chain of invocations

```
p' = p.operator$space(new F()).operator$space(new Left())
    .operator$space(new F()).operator$space(new Right())
    .operator$space(new Right()).operator$space(new F())
    .operator$space(new Left()).operator$space(new F())
```

But node expressions may also be prefixed by the operators

```
>       <       <->     ---     >>      <<      >>>     in      ::
+>      <+      <+>     -+-     >=      <=      <=>     ++      --
/>      </      </>     -/-     +       *       /       %       **
-->     <--     <-->    |       ||      &       &&
```

For each operator, the corresponding operator method is invoked on the current producer (for their names, see Table 6.1; the operator `+` is considered unary here so that the name is `operator$pos`). But a producer class need not implement all operator methods. In particular, they are not part of some interface. If a prefix operator is used for which there is no operator method, this is a compile-time error and indicates that the programmer wanted to use an operator which is not provided by the current producer implementation. As an example for the mechanism of node prefixes, the right-hand side

```
X < Y +> Z
```

is translated to

```
p' = p.operator$space(new X()).operator$lt(new Y())
    .operator$plusArrow(new Z())
```

The implementation of operator methods should be consistent with the meaning of the operator symbols for left-hand sides. E. g., the operator `<` should lead to a reverse successor edge and `+>` to a branch edge. However, this is only a recommendation and not specified by the XL programming language. Note that all prefix operators, when used on right-hand sides, have the same syntactical priority and are applied in the order in which they appear. Thus, while `a > b * c` would be interpreted as `a > (b * c)` in a normal expression context, this does not hold in the context of production statements.

Besides the listed simple prefix operators, there are also the prefix operators

`-`*e*`->` *n* `<-`*e*`-` *n* `-`*e*`-` *n* `<-`*e*`->` *n*

where *e* is an *edge expression*. These become translated to invocations of methods on the current producer *p* as follows:

- If *e* is an identifier (possibly followed by a parenthesized list of arguments) which denotes a name of a method which is a member of the type of *p* and which can be invoked with *n* as first argument and the optional list as remaining arguments, then this method is invoked. The optional list of arguments may start with an additional implicit argument of type `de.grogra.xl.query.EdgeDirection` to which the direction of the operator is passed. For example, if the type *P* of the producer *p* declares the methods

  P `insert(Node node);`
  P `insert(Node node, EdgeDirection dir, int position);`

  then the sequence of production statements

  ```
  -insert-> n -insert(3)-> m
  ```

  is translated to

  p′ `=` *p*`.insert(n).insert(m, EdgeDirection.FORWARD, 3)`

- Otherwise, *e* is treated as a normal expression. Depending on the used operator, one of

  p`.operator$arrow(`*n*`,` *e*`)`
  p`.operator$leftArrow(`*n*`,` *e*`)`
  p`.operator$sub(`*n*`,` *e*`)`
  p`.operator$xLeftRightArrow(`*n*`,` *e*`)`

  is invoked.

As has been said above, the result of an invocation of an operator method on the current producer is taken as the new current producer. A producer may choose to simply return itself. But it may also return some other producer, even of another type. The implementation of a parallel vertex-vertex algebra for the usage in XL programs (see Sect. 4.8.4 on page 85 and Sect. 10.6 on page 335) makes use of this. There, for example, a component

```
a b in v
```

of the right-hand side shall ensure that the vertices `a`, `b` are contained in this order in the list of neighbours of vertex `v`. But incorrect code like `a in v` or `a b c in v` shall be rejected since there is no corresponding operation of vertex-vertex algebras. If we used the same producer for all operator method invocations, the rejection would only be possible at run-time because a compiler would not know in which state the producer is at a specific operator application. However, if the operator methods return a producer which implements exactly those operator methods whose operators are allowed as next prefix operators, incorrect code can already be detected at compile-time. We may see this as a deterministic finite automaton where states correspond to producer types, transitions to operator methods of the types, and where the input is the sequence of production statements. If there is no suitable operator method for the given input in the current state, the input is rejected at compile-time.

6.7.4 Subtrees and Unconnected Parts

Like for queries, the symbols `[` and `]` are used to enclose subtrees, and the symbol `,` is used to separate unconnected parts of the right-hand side. These symbols are not mapped to operator methods, but to special producer methods `producer$push`, `producer$pop` and `producer$separate`. Like for operator methods, a producer need not implement these methods, in which case it is a compile-time error to use corresponding symbols within production statements. It is even not specified by the XL programming language that these symbols stand for subtrees or separations, but within typical implementations, they will have this meaning.

The syntax of the XL programming language ensures that bracket symbols have to appear pairwise and are correctly nested. For an opening bracket, `producer$push` is invoked on the current producer p and `producer$begin` on the result of this invocation. The result of the second invocation is taken as the new current producer for the bracketed part. For each closing bracket, at first `producer$end` is invoked on the producer q which is current at that time, and then `producer$pop` is invoked on p with q as argument, where p is the producer which has been current for the corresponding opening bracket. The result of the latter invocation is taken as the new current producer p' behind the brackets. Thus, the right-hand side `x y [a]` is translated to

```
p = p.operator$space(x).operator$space(y);
q = p.producer$push().producer$begin().operator$space(a);
q.producer$end();
p' = p.producer$pop(q);
```

The handling of the separation symbol `,` is very simple, it just leads to an invocation of `producer$separate` on the current producer. As usual, the result defines the new current producer:

```
p' = p.producer$separate();
```

6.7.5 Code Blocks

When computations for node attributes are necessary, it is not always possible or convenient to specify production statements as a plain sequence of (prefixed) node expressions. It may happen that some computations would have to be repeated, or that computations have to be split into several lines for the sake of readability or because control statements like loops are necessary. For the same reasons, L-systems were extended by imperative programming statements (Sect. 3.10 on page 29) so that code blocks can be specified within right-hand sides.

We adopt the syntax of such L-systems, which is also used by the L+C programming language (Sect. 3.14 on page 33), and allow conventional imperative code blocks everywhere within production statements. They have to be enclosed by curly braces:

```
F(x) ==> {float f = x/3;} F(f) RU(-60) F(f) RU(120) F(f) RU(-60) F(f);
```

In order to be able to access local variables declared in such a block (like `f` in the example) within the rest of the production statements, a code block does not introduce its own private scope, but has the same scope as its enclosing production statement block.

6.7.6 Control Flow Statements

The definition of rules of relational growth grammars (Def. 5.12 on page 107) introduces a mapping which assigns to each match of the left-hand side a right-hand side to use for the actual production. As it has been discussed in Sect. 5.4 on page 105, an application of this feature is plant growth where the number of generated plant segments within a single step of the model depends on the (possibly local) internal state of the plant. In practice, such a mapping is most conveniently and versatilely specified by control flow statements within production statements of the right-hand side. The language of the GROGRA software defines the repetition operator as a loop-like control flow statement, and the L+C programming language uses C++-style control flow statements. The XL programming language takes a similar way: almost any control flow statement is also allowed as a production statement, namely the compound control flow statements **if**, **for**, **do**, **while**, **switch** and **synchronized** and the simple control flow statements **break**, **continue** and **throw**. Compound control flow statements may be prefixed with a label as in `label:while(true) ...`, and after such a label it is also allowed to specify a block of production statements in parentheses as in `label:(F(1) [F(1)] ...)`. Compound control flow statements contain substatements, these can be either further compound control flow statements, or statement blocks in braces, or blocks of production

statements in parentheses. The trailing semicolon of **do** and simple control flow statements has to be omitted.

```
x:X [a:A] y:X ==>> x for (int i = 1; i <= a.energy; i++) ([A]) y [a];

Cycle(n) ==> {X first = null; int i = n;}
loop:
    while (true) (
        current:X
        if (--i == 0) (
            first
            break loop
        ) else if (first == null) {
            first = current;
        }
    );

Bud(type) ==>
    switch (type) (
        case INTERNODE:
            Internode
            break
        case FLOWER:
            Flower
            break
    );
```

The first rule specifies the example production of Sect. 5.4 on page 105. The second (deliberately long-winded) rule replaces a `Cycle`-node with parameter `n` by `n` nodes of type `X` which are arranged in a cycle. We can imagine the effect of control flow statements as an expansion of their contained substatements to production statements without control flow statements. The right-hand side of the second rule with $\texttt{n} = 3$ expands to `first:X X X first` which is indeed a cycle of length 3.

The general mechanism of control flow statements within production statements is their usual mechanism within statement blocks with an additional handling of the current producer. Namely, when using control flow statements, there may be different execution paths by which a given code location can be reached. But within production statements, the current producer has to follow the execution path. Thus, if a statement can be reached by several execution paths, the current producer is given by the current producer at the end of the actually executed path. This means that we have to restrict its compile-time type: the best what we can do is to assume that this type is given by the least upper bound [72] of the producer types at the end of all execution paths which can reach the statement. For an **if**-statement, this means that the type is the least upper bound of the producer types at the end of the **if**- and **else**-branches. For loop statements, their body may be executed repeatedly, so the producer which is current at the end of the body may become the

current producer at the beginning of the next repetition. In order for this to be possible, the type of the current producer at the end has to be assignable to the type of the current producer at the beginning. The same holds for the type at **continue**-statements. Furthermore, the body of **for**- and **while**-loops may not be executed at all. Together with the previous consideration, this forces us to assume that the type of the current producer after such a loop is the same as before the loop. Contrary, **do**-loops execute their body at least once so that the type of the current producer after such a loop is the same as at the end of its body. Statements can also be reached by **break**-statements, then we have to take into account the type of the current producer when computing the least upper bound of producer types at the end of the target of the **break**-statement. Within **switch**-statements, the least upper bound has to be computed for a statement group whose preceding statement group falls through the label. For other statement groups, the current producer is the current producer at the beginning of the **switch**-statement. **synchronized** and **throw** do not require a special treatment.

6.8 Rules

The previous sections presented the two main building blocks for rules: queries for left-hand sides and production statements (defined on top of operator overloading) for right-hand sides. In order to combine queries and production statements to rules, the XL programming language defines two arrow symbols `==>>` and `==>`, for which there were already some examples in the previous section. It is not specified how their rules differ in semantics, this depends on the implementation of the used producer. In our examples, we assume that rules using `==>` extend rules using `==>>` by implicit connection transformations according to Def. 5.6 on page 100. I. e., with `==>` we can write L-system rules just as we would do this in an L-system environment.

A third rule arrow `::>` differs from the other two in that its right-hand side is a single imperative statement. Such an *execution rule* executes the statement for every match, in contrast to the previous *structural rules* whose right-hand sides consist of production statements which define a successor graph. For example, the execution rules

```
x:F & (x.diameter > 0.01) ::> println(x);
c:Cylinder ::> {
    println(c);
    println("  Radius = " + c.getRadius());
    println("  Length = " + c.getLength());
}
```

simply print some information about each match of their left-hand side. Execution rules are also useful if we do not want to modify the topology of the current graph, but only its internal state, i. e., attribute values of nodes.

This typically happens for functional plant models which, e. g., compute new concentration values of nutrients, distribute produced carbon among plant entities, or have some transport mechanism for mobile substances. Note that in most cases, execution rules may be substituted by **for** loops as in

```
for ((* x:F & (x.diameter > 0.01) *)) {
    println(x);
}
```

The difference is that execution rules implicitly provide a current producer, see Sect. 6.8.2 on the facing page.

6.8.1 Rule Blocks

Rules are specified in *rule blocks*. While conventional imperative blocks of statements use curly braces as delimiters, rule blocks use square brackets. They can appear almost everywhere where a statement block may appear, namely as whole method bodies and as individual statements. So it is possible to write

```
1  void derive() [
2      Axiom ==> F Right Right F Right Right F;
3      F ==> F Left F Right Right F Left F;
4  ]
5
6  void update() {
7      if ((++time % CONCENTRATION_DELTA) == 0) [
8          x:Substance ::> x.updateConcentration();
9      ]
10 }
```

A rule block is sequentially executed in textual order like a normal block. Thus, the rule in line 3 is executed *after* the execution of the the rule in line 2 has finished. This may seem to contradict the intended parallel mode of application of relational growth grammars, but in fact, it does not. For the base implementation of the XL interfaces (Sect. 9.1 on page 235), it is only the parallel production of the two-level RGG derivation which is constructed sequentially, its final application is nevertheless a parallel derivation.

Simple rule blocks as the above examples use the same mechanism as query expressions to determine the current graph implicitly (see Sect. 6.5.13 on page 158). In order to specify the current graph explicitly, we again use the mechanism of query expressions for the current graph and its compile-time model, namely we prepend the current graph in front of a rule block as in

```
graph.[
    Sphere ==> Box;
]
```

Such a statement can appear everywhere where individual statements are allowed, but not as an entire method body.

Besides rules, also normal statement blocks in braces can be inserted in rule blocks. Thus, we can arbitrarily nest statement blocks and rule blocks.

6.8.2 Execution of Rules

The execution of a rule starts with the execution of the query of the left-hand side. For each match of the query pattern which is found by the algorithm described in Chap. 7, the implementation of the used graph provides a producer p. This has to be an instance of the subinterface of `de.grogra.xl.query.Producer` which is specified by the compile-time model, see the class diagram in Fig. 6.2. Then, the method `producer$beginExecution` is invoked on p, where the argument passed to `arrow` indicates the arrow symbol of the rule: the constant `SIMPLE_ARROW` is used for `==>`, `DOUBLE_ARROW` for `==>>` and `EXECUTION_ARROW` for `::>`. If this invocation returns **false**, the right-hand side is not executed, and the method `producer$endExecution` is invoked immediately with **false** as argument. Otherwise, the right-hand side is executed with p as initial current producer, and `producer$endExecution` is invoked afterwards on p with **true** as argument. This completes the processing of the current match, and the control flow returns to the query in order to find the next matches.

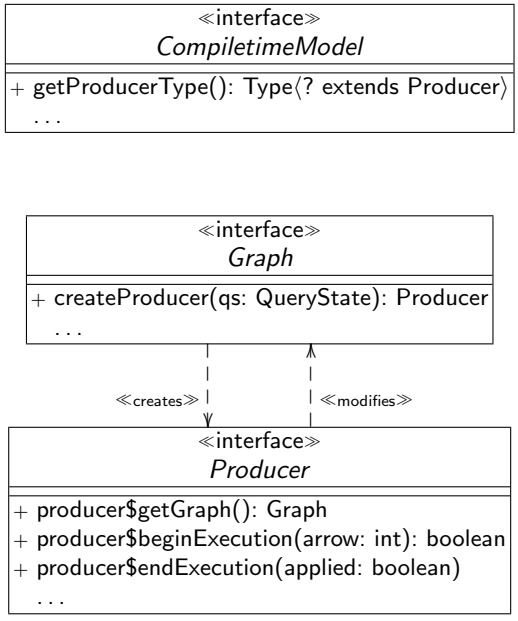


Figure 6.2. Class diagram for producers of rules

The mechanism with a **boolean** result of `producer$beginExecution` which controls whether the right-hand side is actually executed is useful for the implementation of a sequential nondeterministic mode of derivation, see Sect. 9.1.5 on page 245.

6.9 Stand-Alone Production Statements

Production statements can also be composed to single statements within imperative code. The syntax for such *stand-alone production statements* is

`==>` *production statements*`;`
p `==>` *production statements*`;`

where *p* is an expression. The initial current producer for the production statements of the first variant is given by the current producer of the innermost enclosing production statement. If this does not exist, a compile-time error occurs. For the second variant, the initial current producer is the result of the expression *p*. Note that the syntax is similar to rules, but stand-alone production statements occur as single statements or within normal statement blocks, while rules occur within rule blocks. For an example, see Sect. 6.7 on page 162.

6.10 Properties

The XL programming language defines *properties* of objects which have a similar meaning as instance fields. However, they are addressed by a different syntax, and additional colon-prefixed assignment operators exist. Contrary to fields, the XL programming language does not specify a means to introduce new properties, this is up to the implementation of a data model. Properties are addressed by the syntax

e`[`*n*`]`

where *e* is an expression of reference type *T* and *n* is the name of a property declared in *T*. The property access *e*`[`*n*`]` stands for a variable whose type is the type of the property, and whose value is read and written by special methods provided by a run-time model. The details are described in the following.

6.10.1 Compile-Time and Run-Time Models for Properties

Like graphs, also properties are defined by a data model which consists of a compile-time model and a run-time model. These define which properties exist and how their values are read and written. Thus, while a data model for graphs is concerned with the topology, a model for properties is concerned with internal state (e. g., attribute values).

The interface `de.grogra.xl.property.CompiletimeModel` together with its nested interface `Property` represent static compile-time aspects of properties. For an expression e`[`n`]` where the type of e is T, it is checked if the current scope has an enclosing declaration with an annotation of type `@de.grogra.xl.property.UseModel` whose `type` element specifies a supertype of T or T itself. If this is the case, an instance of the type specified by the `model` element is used as the `CompiletimeModel` instance. Otherwise, the type T has to have or inherit an annotation `@de.grogra.xl.property.HasModel` whose single element defines the type to use for the `CompiletimeModel` instance. The first possibility is used by the example

```
class Node {...}
class Label extends Node {...}

@UseModel(type=Node.class, model=MyCompiletimeModel.class)
class Test {
    void test(Label n) {
        System.out.println(n[text]);
    }
}
```

while the following example makes use of the second possibility:

```
@HasModel(MyCompiletimeModel.class)
class Node {...}

class Label extends Node {...}

class Test {
    void test(Label n) {
        System.out.println(n[text]);
    }
}
```

In both cases, the property access `n[text]` uses `MyCompiletimeModel` as compile-time model.

The class diagram of the interfaces for properties is shown in Fig. 6.3 on the following page. The method `getDirectProperty` of the compile-time model is invoked at compile-time with the arguments (T, n) to obtain the corresponding compile-time representation of the property. For the example, T would be `Label` and n would be `text`. If this invocation returns **null**, no property named n is declared in T, and a compile-time error occurs. Otherwise, the method `getType` of a property determines its type, i. e., the type of the variable e`[`n`]`, and `getRuntimeType` determines the type of the run-time representation of the property. The latter has to be a subtype of the interface `RuntimeModel.Property` which declares getter- and setter-methods to read and write values of property variables. Run-time models for properties are obtained in a similar way as run-time models for graphs: the run-time name of the compile-time model (as reported by `getRuntimeName`) is used

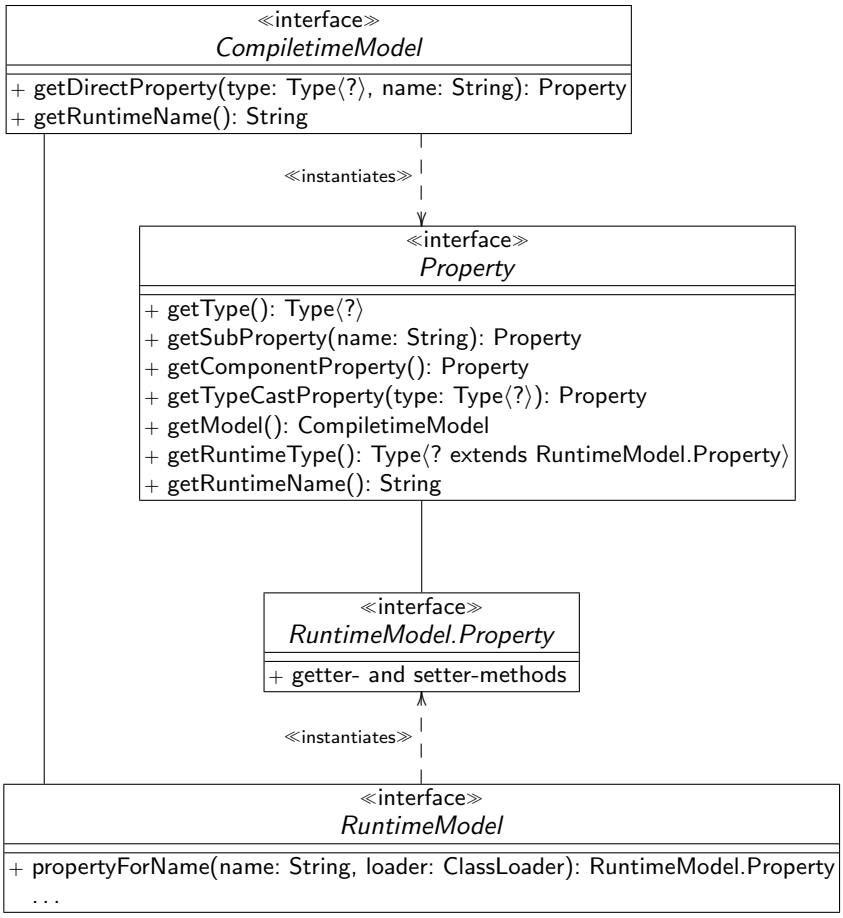


Figure 6.3. Class diagram of compile-time and run-time models for properties

at run-time as input to a `de.grogra.xl.property.RuntimeModelFactory` in order to obtain the corresponding `RuntimeModel`. The run-time name of the compile-time `Property` is then passed to the `propertyForName` method of the `RuntimeModel` to get the run-time representation p of the `Property`.

6.10.2 Access to Property Variables

A property access expression e`[`n`]` addresses a property variable which associates a value for property n with the instance that results from e. How the value is associated is up to the implementation of the run-time property p. Namely, the value is read by invocation of the getter-method of

`RuntimeModel.Property` which is suitable for the property type, and it is written by invocation of the suitable setter-method. It is not specified by the XL programming language that these methods are consistent, i. e., that a getter-method returns the value which has been set by the last invocation of a setter-method, although reasonable implementations will usually do so. Unlike access to field variables, the method implementations may have side-effects. E. g., writing a value may trigger the invocation of notification methods of listeners which are associated with the property variable.

A property access may be concatenated as in `sphere[position][z]` if a property declares subproperties itself; the declaration is done by the method `getSubProperty` of the compile-time representation of the property. This may be combined with type casts as in `((Matrix4d) group[transform])[m00]`, see the method `getTypeCastProperty`. Furthermore, if the type of a property is an array type or a subtype of `java.util.List`, the property may declare component properties by the method `getComponentProperty` which are addressed by an indexed access as in `container[children][2][position]`. Property access using such a concatenation of properties is still considered as a single property access with the property being a compound property. This means that while the expression `container.children[2].position` which uses conventional field and array access is equivalent to the expression `((container.children)[2]).position`, this does not hold for property access: `container[children][2][position]` denotes the compound property `[children][2][position]` of `container`, while the parenthesized expression `((container[children])[2])[position]` denotes the property `position` of the element at index 2 of the value of the property `children` of `container`. While both variants should return the same value for reasonable implementations, there may be different side-effects when values are written to these properties: in the first case, the `container` itself could be notified about the modification, in the second case the element with index 2 of the `children` of the `container`.

6.10.3 Deferred Assignments

Like for any variable of the XL programming language, values may be assigned to property variables by the assignment operator `=`. Also compound assignment operators like `+=` are defined which at first obtain the current value of the variable, then perform some operation on it and finally write the result to the variable. Specifically for the usage with properties, the colon-prefixed assignment operators

```
:=      :**=    :*=     :/=     :%=     :+=     :-=
:<<=    :>>=    :>>>=   :&=     :^=     :|=
```

are introduced (but by overloading they may also be used for other purposes). Although the XL programming language does not specify the semantics of implementations of these operator methods, the convention is to use them for

deferred assignments which are not executed immediately, but at some later point. Within the context of relational growth grammars, these assignments are deferred until the application of a parallel production is executed. Simple deferred assignments correspond to assignments of values to node attributes, compound deferred assignments to incremental modifications of attribute values (see Sect. 5.8 on page 115).

Deferred assignments help to solve a problem which was described by Robert Floyd in his famous Turing Award Lecture [58]. As he points out, the natural way to implement the update of some state vector is to just list the individual update equations for the components. But, given the sequential execution of the imperative programming paradigm, this natural way leads to incorrect results: for the update equation of a component, it erroneously uses the modified values for those components that have already been updated. As an example, consider the Game of Life (Sect. 2.4 on page 14), whose cell states can be regarded as a state vector. The transition rule which lets dead cells (state 0) come to life (state 1) can be implemented as

```
x:Cell & (x[state] == 0) & (sum((* x -neighbour-> Cell *)[state]) == 3)
    ::> x[state] := 1;
```

If we used the assignment `=` instead of `:=`, cells would immediately come to life within the ongoing execution of the rule for the current step. This then influences the application conditions of the left-hand side for the remaining matches of `x:Cell`, so that the overall behaviour depends on the order in which these matches are found and does not conform to the Game of Life. The usage of `:=` as assignment operator defers the actual assignment of the new state to the final application of the parallel production. This correctly models the parallel mode of transitions of cellular automata, and it correlates with the parallel application of rules. In general, the immediate assignment of new values conflicts with parallel derivations.

The internal mechanism of deferred assignments is as follows: if the left-hand side of a deferred assignment operator denotes a property variable and the right-hand side some expression, it is tried to apply the operator as a quaternary operator to the arguments consisting of the run-time representation p of the property, the object to which the property variable belongs, an **int**-array of indices of component properties (having a length of zero if no component property is used), and the value of the right-hand side. For example, for the assignment `x[value] :+= 1` the arguments are (p, `x`, **new int**`[]{}`, 1) where p denotes the run-time representation of the property named `value`. Then the usual resolution of operator overloading as described in Sect. 6.6 on page 160 is applied (see Table 6.1 for the translation from symbols to operator method names). For the above example and if p declares a suitable operator method on its own, the final translation of the example is p`.operator$defAddAssign(x, new int[]{}, 1)` A translation of the assignment `x[children][2][position][z] := 3` is q`.operator$defAssign(x, new int[]{2}, 3)` where q denotes the run-time

representation of the subproperty `z` of the subproperty `position` of the component property of the property `children` of the type of `x`. The validity of the index arrays is not guaranteed to persist after returning from the operator method. Therefore, a compiler is free to choose other equivalent, but more efficient ways to allocate the required **int**-arrays, for example by using some pool of **int**-arrays instead of allocating a new array for each deferred assignment.

6.10.4 Properties of Wrapper Types

The compile-time model of a graph (Sect. 6.5.1 on page 142) specifies the method `getWrapProperty` which takes the type of a wrapper node (Sect. 6.5.2 on page 148) as input and returns the property which contains the wrapped value (or **null** if no such property exists). E. g., a node class `ObjectNode` wrapping `Object`-values might declare a property `value` for the wrapped values. Now consider a rule

```
v:Vector3d ::> v[z] = 0;
```

which shall set all z-components of vectors to zero. If `Vector3d` cannot be used as node class, it has to be wrapped in an `ObjectNode` so that the pattern actually looks for instances of `ObjectNode` which wrap a value of class `Vector3d`. Now assume that the used implementation of properties only defines properties with respect to node types, i. e., the non-node class `Vector3d` has no property `z`, but each `Vector3d`-valued property of a node type has a subproperty `z` (this is the case for the implementation of GroIMP, see Appendix B.10). Then the above rule has no direct meaning, but has to be interpreted to set the subproperty `z` of the type-cast property to `Vector3d` of the property `value` of each `ObjectNode` wrapping a `Vector3d` to zero. I. e., `v` is implicitly replaced by `((Vector3d) $v[value])` with `$v` standing for the wrapper node (Sect. 6.5.2 on page 148).

6.11 Module Declarations

Parametric L-systems define the notion of *modules*, i. e., parameterized symbols (Sect. 3.7 on page 25). In our setting, the natural representation of such symbols is as nodes of classes which declare a single instance field for every parameter. For example, to model an L-system rule $\mathsf{X}(a) \rightarrow \mathsf{X}(a+1)$, we would declare a node class like

```
class X extends Node {
    float a;

    X(float a) {
        this.a = a;
    }
}
```

and write `X(a) ==> X(a+1);`. The class declaration suffices for the right-hand side which creates a new `X` node by invocation of the constructor, but it does not provide a user-defined pattern (see 6.5.2) for the pattern `X(a)` of the left-hand side – there is not yet a meaning what `X(a)` stands for as a pattern. One could define that in this situation (and analogous ones), where the argument `a` of a parameterized pattern `X(a)` denotes the name `a` of a field declared in the node type `X` of the pattern, the compiler shall assume a simple node pattern for the node type `X`, declare a query variable `a` and bind the value of the field `a` for matched `X` nodes to this query variable. But this conflicts with traditional L-systems which access parameters by position only so that we may write `X(a)` in one rule and `X(b)` in another, with both `a` and `b` being bound to the same parameter. If we really want to access parameters by name, we have to write `x:X ==> X(x.a+1);`, but to access parameters by position, the declaration of a user-defined pattern by a subclass of `de.grogra.xl.query.UserDefinedPattern` is mandatory. As this requires a relatively large amount of coding, the XL programming language defines *module declarations* which automatically declare a suitable user-defined pattern, but otherwise are like normal class declarations.

6.11.1 Syntax

The basic syntax of module declarations is similar to the L+C programming language (Sect. 3.14 on page 33). If there is a single parameter `a` of type **float**, we write

```
module X(float a);
```

This is equivalent to a class declaration

```
class X extends N {
    float a;

    X(float a) {this.a = a;}

    public static class Pattern extends UserDefinedPattern {
        private static void signature(@In @Out X node, float a) {}
        ... // suitable implementation of abstract methods
    }
}
```

The declaration uses an implicit superclass N which is determined by the annotation `@de.grogra.xl.modules.DefaultModuleSuperclass` of the innermost enclosing declaration which has such an annotation. If such an annotation does not exist, `Object` is used as superclass.

Module declarations may also specify their superclass explicitly, use inherited fields in the pattern, and contain an entire class body:

```
module Y(super.a, String b) extends X {
```

```
    public String toString () {return "Y[" + a + "," + b + "]";}
}
```

is equivalent to (given the previous declaration of `X`)

```
class Y extends X {
    String b;

    Y(float a, String b) {super(a); this.b = b;}

    public static class Pattern extends UserDefinedPattern {
        private static void signature(@In @Out Y node,
                                      float a, String b) {}
        ... // suitable implementation of abstract methods
    }

    public String toString () {return "Y[" + a + "," + b + "]";}
}
```

It is also possible to declare the superclass constructor explicitly, to add a list of expression statements to be included in the constructor (using the syntax of with-instance expression lists, Sect. 6.13.4 on page 187), and to specify a list of implemented interfaces:

```
module Z(super.a) extends Y(a, "Z").(a2 = a*a, System.out.println(this))
    implements I {
    float a2;
}
```

is equivalent to (given the previous declaration of `Y`)

```
class Z extends Y implements I {
    float a2;

    Z(float a) {super(a, "Z"); a2 = a*a; System.out.println(this);}

    public static class Pattern extends UserDefinedPattern {
        private static void signature(@In @Out Z node, float a) {}
        ... // suitable implementation of abstract methods
    }
}
```

A further special syntax is possible within module declarations. To be able to deal with situations where we want to use a field of a superclass as a parameter, but do only have access to this field via get- and set-methods (due to restricted visibility or even because the value is not stored directly in a field, but by some other mechanism), we have to specify the get-method in the declaration of the parameter:

```
module W(float a return getA()) extends X.(setA(a));
```

Note that this example assumes the existence of the methods `getA` and `setA` in `X`, which is not the case for the definition of `X` from above.

6.11.2 Instantiation Rules

Module declarations support one distinguished feature, namely the declaration of *instantiation rules*. While all other features of the XL programming language are not related to a specific field of application, instantiation rules are of best use within the context of (2D and) 3D applications. Instantiation in the context of the representation of geometry refers to the multiple inclusion of the same geometric structure but with different locations in space [59]. Thus, we need only one representation of the *master* (e. g., a detailed description of the 3D geometry of a tree), although it is displayed several times. This of course helps to reduce memory consumption of the whole description. A more general concept of instantiation is to algorithmically create complex geometry out of a compact specification "on the fly", i. e., when the actual geometry has to be displayed or otherwise accessed [121, 37]. For example, the specification could describe a set of parameters for a tree-construction algorithm, the algorithm then creates the tree on demand without persistently storing the geometry in memory.

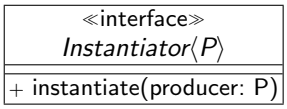
Instantiation rules of modules can be used for both kinds of instantiation. They consist of production statements (Sect. 6.7 on page 162) and are specified with a leading arrow `==>` as last part of a module declaration:

```
module X {
    const Sphere sphere = new Sphere();
} ==> sphere;

module Y(int n)
  ==> for(1 : n)(Cylinder.(setShader(RED)) Cylinder.(setShader(WHITE)));
```

The first example creates a single sphere as master and instantiates this (fixed) sphere when needed. The second example represents a simple geometry-creating algorithm with a parameter `n`: it instantiates a sequence of 2`n` cylinders with alternating colours.

Instantiation rules can be used for similar purposes as interpretive productions of L-systems (Sect. 3.9 on page 28). Both assign additional structure to objects which the objects do not have on their own. This may be used, e. g., to define a geometric view of objects without having to deal with the geometry when specifying the behaviour of the objects themselves. The difference between instantiation rules and interpretive productions is that the former do not modify the persistent structure (the graph), but only create temporary structure on demand, while the latter insert additional structure into the persistent structure and remove the additional structure at a later point. Concerning space and time efficiency and principles of clean software design, instantiation rules are preferable, but they cannot have an effect on objects in the persistent structure which is sometimes used by interpretive productions.

The internal mechanism of instantiation rules makes use of the interface `de.grogra.xl.modules.Instantiator`:



For a declaration of an instantiation rule `==>` *production statements*`;`, the annotation `@de.grogra.xl.modules.InstantiationProducerType` of the innermost enclosing declaration which has such an annotation defines the type P of the producer for the instantiation. The module class then implicitly implements the interface `Instantiator<`P`>`. If the module does not declare parameters, this looks as follows:

```
public void instantiate(P producer) {
    producer ==> production statements;
}
```

Otherwise, if there are parameters, an additional method is declared. In case of a single parameter **float** `a`, this looks like

```
public void instantiate(P producer) {
    instantiate(producer, a);
}

public void instantiate(P producer, float a) {
    producer ==> production statements;
}
```

So the body of the method `instantiate` consists of a single stand-alone production statement which uses the parameter `producer` as initial current producer and which contains the production statements of the instantiation rule. (Thus, instantiation rules are nothing but useful "syntactic sugar".) It is up to the application which uses instances of modules with instantiation rules to provide a suitable implementation of the producer, and to invoke the `instantiate` method when needed.

6.12 User-Defined Conversions

The Java programming language defines automatic boxing and unboxing conversions from primitive types to their wrapper classes and reverse. I. e., in

```
int i = 0;
Integer j = i;
i = j;
```

the methods `Integer.valueOf(int)` and `Integer.intValue()` are invoked implicitly to convert between the otherwise incompatible types `Integer` and **int**. However, unlike the C++ and C# programming languages [83, 41], the Java programming language does not allow *user-defined conversions*. These are of particular use in combination with operator overloading to write expressions

in a natural, uncluttered way. The C++ programming language allows conversions by constructors and by conversion functions, which are declared as special conversion operator functions, while the C# programming language only allows conversions by conversion functions. The XL programming language follows the C++ programming language and allows user-defined conversions by constructors and conversion functions, but conversion functions are not declared by operator methods. Instead of this, conversion functions are declared by methods which conform to the following patterns often found in APIs for the Java programming language:

```
class X {
    X(S source);
    static X valueOf(S source);
    T toT_s();
    t tValue();
}
```

where X, S are types, T a reference type with T_s its simple name, and t a primitive type. I. e., a type X may declare conversions from a source type S to itself by a constructor or a static `valueOf` method, and it may declare conversions from itself to target types T, t by instance methods `to`T_s, t`Value`, respectively. To be able to declare user-defined conversions between types which do not know each other, it is also possible to declare static conversion methods from S to T in some other class X:

```
class X {
    static T toT_s(S source);
}
```

But then the method `to`T_s has to be statically imported so that it can be found.

While the names `valueOf` and `to`T_s for methods indicate that the methods were specifically designed for conversion purposes, constructors do not always convert values into their representation of another type. For example, the class `StringBuffer` declares a constructor with a single **int** parameter, its initial capacity. So an assignment `StringBuffer b = 42;` does not create a buffer with the character sequence `"42"` as its content, but an empty buffer with initial capacity 42. So in order to prevent programming errors, user-defined conversions by constructors are disabled by default except for constructors which are annotated with `@de.grogra.xl.lang.ConversionConstructor` as in

```
class Int {
    int value;

    @ConversionConstructor
    Int(int value) {this.value = value};
}
...
```

```
Int i = 10; // uses conversion constructor
```

Allowed conversions within source code can be controlled by the annotation `@de.grogra.xl.lang.UseConversions` which specifies a list of a subset of the values `VALUE_OF`, `TO_TYPE`, `TO_TYPE_IN_SCOPE`, `CONVERSION_CONSTRUCTOR`, `CONSTRUCTOR` of the enumeration `de.grogra.xl.lang.ConversionType`:

```
@UseConversions({VALUE_OF, TO_TYPE, CONVERSION_CONSTRUCTOR, CONSTRUCTOR})
class Test {
    StringBuffer buf = 42; // creates a StringBuffer of capacity 42
}
```

Besides user-defined conversions, also implicit conversions from **double** to **float** are possible, see Sect. 6.13.2 on the following page.

6.13 Minor Extensions

This section presents the minor extensions of the XL programming language with respect to the Java programming language. It concludes with a list of all defined operators.

6.13.1 for statement

The Java programming language defines the basic **for**-statement as it is known from the C programming language and an *enhanced* **for**-statement with the syntax

for (*T i* : *e*) *b*

where i is the identifier of a local iteration variable to be declared for the loop and T its type, e is an expression and b the body of the loop. For the Java programming language, the type of e has to be either an array type in which case i iterates over the values of the elements of the array, or the type of e has to be a subtype of `java.lang.Iterable` in which case i iterates over the values of the iterator returned by `Iterable.iterator()`.

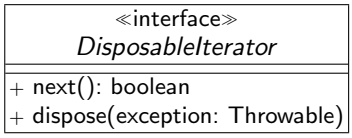
The XL programming language extends the syntax by also allowing enhanced **for**-statements without an iteration variable:

for (*e*) *b*

While this is useless for iterations over arrays, it may be useful for iterations over an `Iterable` if we are interested in side-effects of the iteration only. But it is most useful for the two new types of iterations defined by the XL programming language. First, an enhanced **for**-statement may iterate over the values of e. If e is a normal expression with a single value, this is not very useful, but if e is a generator expression, the iteration is over every yielded value of e. Using this type of iteration, we may write

```
for (int i : 0 : 100) {...}
for (Shoot s : (* Shoot *)) {...}
for ((* s:Shoot *)) {...}
```

A second new usage of the **for**-statement is by expressions whose type I is a subtype of the new interface `de.grogra.xl.lang.DisposableIterator`. This interface is declared as follows:



If the type I also has an instance method `value()`, the iteration over such a *disposable iterator* is equivalent to the code

```
I #it = e;
Throwable #t = null;
try {
    while (#it.next()) {
        T i = #it.value();
        b
    }
}
catch (Throwable #u) {
    #t = #u;
    throw #u;
}
finally {
    #it.dispose(#t);
}
```

So the `next` method governs the loop, the invocation of `value` obtains the value for the current iteration, and the final invocation of `dispose` is guaranteed even in case of an exception. If I does not have a method `value()`, the disposable iterator iterates over **void**, then an enhanced **for**-statement without iteration variable has to be used. Such a **for**-statement can be used as a counterpart of the **using**-statement of the C# programming language which obtains a resource, then executes some statements and finally invokes `Dispose` on the resource to indicate that the resource is no longer needed [41].

6.13.2 Implicit Conversions from double to float

Conversion from **double** to **float** are narrowing conversions and, thus, have to be specified explicitly by a cast to **float** in the Java programming language. For convenience, in the XL programming language it is possible to enable implicit conversions from **double** to **float** where needed. This is done by the annotation `@de.grogra.xl.lang.ImplicitDoubleToFloat` whose single **boolean** element controls whether conversions from **double** to **float** shall be implicit within the annotated entity or not.

```
@ImplicitDoubleToFloat // default value is true
class Test {
    float x = 4.2;     // OK, implicit narrowing conversion

    @ImplicitDoubleToFloat(false)
    void test() {
        x = 4.2;       // compile-time error
    }
}
```

6.13.3 Expression Lists

The C programming language and most programming languages which syntactically follow the C programming language define the *comma operator* a`,` b which evaluates a, discards the result, and then evaluates b and returns the result of the latter. The Java programming language does not define such an operator (with the exception of statement lists for initializer and update statements of **for**-loops). The XL programming language does not specify the comma as an operator, too, but there are places where the Java programming language only allows a single expression, while the XL programming language allows a comma-separated *expression list* which may even declare local variables whose scope is the rest of the expression list. Most important, such a list can be used within parentheses:

```
(int x = 0 : 100, x * x)
```

The usage of an expression list is helpful in conjunction with generator expressions if there is a need for some intermediate computations. For example, assume that we want to analyse the branching angles within a plant structure composed of nodes of type `Shoot`. The query `(* a:Shoot [b:Shoot] *)` finds all branching locations with the parent shoot `a` and the branching shoot `b`. Given a suitable method `angle` and a suitable aggregate method `mean`, the complete expression is

```
mean(((* a:Shoot [b:Shoot] *), angle(a, b)))
```

6.13.4 With-Instance Expression Lists

When several methods or fields of the same object have to be addressed, *with-instance expression lists* are useful. Their syntax is

i`.(`e_1`, ..., `e_n`)`

where i is an expression of reference type and $e_1, \ldots, e_n$ are expressions. Within the parentheses, fields or methods which are members of the type of i are in scope, i. e., they can be referred to by their simple name. If an instance field or instance method is addressed, the result of i is taken as the

target reference. This is similar to the implicit **this** within bodies of instance methods. With-instance expression lists can be compared to **with**-statements of the Pascal programming language. The result of a with-instance expression list is the result of i. For convenience, the result of i can be addressed within the expression list by the identifier `$`.

A typical case where one can benefit from with-instance expression lists is the creation of a new object with a lot of attributes. Its class could provide a constructor which configures all attributes, but the list of constructor parameters then is quite complex, and one has to remember the order of parameters since one cannot use their name within an invocation:

```
panel.add(new JLabel(t, i, a));
```

Furthermore, the Java programming language does not define invocations with default arguments, so we either would have to specify arguments for every parameter, or the implementation of the class would have to declare a lot of constructors with different sets of parameters and default values for the remaining attributes. The alternative to complex constructors is to use an invocation of a setter-method for each attribute. This is more flexible and readable, but leads to unnecessarily lengthy code because we have to introduce a temporary variable which holds the object to be configured:

```
JLabel lbl = new JLabel();
lbl.setText(t);
lbl.setIcon(i);
lbl.setHorizontalAlignment(a);
panel.add(lbl);
```

With the help of with-instance expression lists, this can be solved in an easy way:

```
panel.add(new JLabel().(setText(t), setIcon(i),
                        setHorizontalAlignment(a)));
```

6.13.5 Anonymous Function Expressions

The Java programming language defines anonymous class declarations as a syntactical shorthand. However, when such a declaration is only needed to declare a single method as its member, there is still some amount of textual overhead. For example, to implement a `de.grogra.xl.lang.DoubleToDouble` function, one has to write

```
DoubleToDouble f = new DoubleToDouble() {
    public double evaluateDouble(double x) {
        return x * Math.sin(x);
    }
}
```

Because the specification of functions like this is often necessary within models which contain numerical computations (e. g., parameter fitting), the XL programming language provides a shorthand for *anonymous function and generator* expressions. The expressions

X x `=>` Y e
X x `=>` Y`*` e

return new instances of the interfaces X'`To`Y' or X'`To`Y'`Generator`, respectively, where X, Y are types, x is an identifier which can be accessed within the expression e as a local variable of type X, and the type of e has to be assignable to Y. For primitive types, X' is the name of the type with a capitalized first letter (e. g., `Int` for **int**). For reference types, it is `Object`, and the type X is appended to the list of type parameters of the interface. The single method of the interface is implemented by returning or yielding the value of e. I. e., the above example can equivalently be written as

```
DoubleToDouble f = double x => double x * Math.sin(x);
```

and the anonymous generator declaration

```
ObjectToObjectGenerator<Node,Shoot> children
    = Node parent => Shoot (* parent --> Shoot *);
```

is a shorthand for

```
ObjectToObjectGenerator<Node,Shoot> children
    = new ObjectToObjectGenerator<Node,Shoot>() {
    public void evaluateObject (ObjectConsumer<? super Shoot> cons,
                                Node parent) {
        yield (* parent --> Shoot *);
    }
}
```

The C# programming language defines anonymous method expressions [41] and, in its latest extension which is not yet a standard, anonymous function expressions, also called λ-expressions due to their similarity with the λ-calculus [128]. The latter have a syntax similar to the presented one.

6.13.6 const modifier

In the Java programming language, constants can be defined by field declarations with both modifiers **static** and **final**. As a simplification, the new modifier **const** can be used for this purpose in the XL programming language, i. e., we may define constants like

```
const int N = 100;
const Shader leafShader = shader("leaf");
```

6.13.7 New Operators

The XL programming language defines several new operators. The operators a `:` b, a `[:]`, a `::` b and a `in` b as well as colon-prefixed assignment operators have already been presented. There are also the arrow operators a `->` b, a `<-` b, a `<->` b, a `-->` b, a `<--` b, a `<-->` b, a `---` b, a `+>` b, a `<+` b, a `<+>` b, a `-+-` b, a `/>` b, a `</` b, a `</>` b, a `-/-` b whose main purpose is to be used within production statements and which have no built-in meaning, i. e., they always refer to operator methods. The new operators a `**` b, a `<=>` b and `` `a` `` have a built-in semantics which is described in the following. A complete list of all operators including their precedence is given in Table 6.1.

Exponentiation Operator

The binary *exponentiation operator*

a `**` b

is defined for operands of type **float** or **double** and computes the value a raised to the power of b. The computation is implemented by the method `pow` of `java.lang.Math`.

Comparison Operator

The binary *comparison operator*

a `<=>` b

is defined for operands of numeric type. Its result type is **int**, and the value is 0 if a `==` b, 1 if a `>` b and -1 otherwise. The last case includes a `<` b, but also cases where a or b is Not-a-Number.

Quote Operator

The unary *quote operator*

`` `a` ``

is defined for operands of any type and performs the identity operation, i. e., type and value are determined by the operand a. Thus, it is the same as `(`a`)`, but the quote operator may be overloaded. This operator can be used for node patterns and node expressions if expressions have to be parenthesized, but normal parentheses cannot be used for syntactical reasons.

List of all Operators

Table 6.1 lists all operators of the XL programming language in order of precedence, starting with the operators with highest precedence. The operators are grouped by horizontal lines, operators within one group share the same precedence. The exponentiation operator `**` and all assignment operators are syntactically right-associative (e. g., `a ** b ** c` means `a ** (b ** c)`), all other binary operators are left-associative. The rules for precedence and associativity follow the Java, C and Perl programming languages.

Table 6.1. Operators of the XL programming language, sorted by precedence

Operator	Description	Operator method suffix	Explanation
`` `a` ``	Quote	`quote`	Sect. 6.13.7
a`[`*b*`]`	Array access, property access	`index`	Sect. 6.10
a`(`*b*`)`	Invocation	`invoke`	Sect. 6.6
a`[:]`	Array generator	`generator`	Sect. 6.3.3
a `->` *b*	Arrow	`arrow`	
a `<-` *b*	Left arrow	`leftArrow`	
a`++`	Postfix increment	`postInc`	
a`--`	Postfix decrement	`postDec`	
a `**` *b*	Exponentiation	`pow`	Sect. 6.13.7
`++`*a*	Prefix increment	`inc`	
`--`*a*	Prefix decrement	`dec`	
`+`*a*	Unary plus	`pos`	
`-`*a*	Negation	`neg`	
`~`*a*	Bitwise complement	`com`	
`!`*a*	Logical complement	`not`	
a `*` *b*	Multiplication	`mul`	
a `/` *b*	Division	`div`	
a `%` *b*	Remainder	`rem`	
a `+` *b*	Addition	`add`	
a `-` *b*	Subtraction	`sub`	
a `<<` *b*	Left shift	`shl`	
a `>>` *b*	Right shift	`shr`	
a `>>>` *b*	Unsigned right shift	`ushr`	
a **instanceof** *T*	Type comparison	-	
a `<` *b*	Less than	`lt`	
a `>` *b*	Greater than	`gt`	
a `<=` *b*	Less than or equal	`le`	
a `>=` *b*	Greater than or equal	`ge`	
a `<=>` *b*	Comparison	`cmp`	Sect. 6.13.7
a **in** *b*	Containment	`in`	Sect. 6.4.1
a `<->` *b*	Left-right arrow	`leftRightArrow`	
a `-->` *b*	Long arrow	`longArrow`	
a `<--` *b*	Long left arrow	`longLeftArrow`	
a `<-->` *b*	Long left-right arrow	`longLeftRightArrow`	
a `---` *b*	Line	`line`	
a `+>` *b*	Plus arrow	`plusArrow`	
a `<+` *b*	Plus left arrow	`plusLeftArrow`	
a `<+>` *b*	Plus left-right arrow	`plusLeftRightArrow`	
a `-+-` *b*	Plus line	`plusLine`	
a `/>` *b*	Slash arrow	`slashArrow`	
a `</` *b*	Slash left arrow	`slashLeftArrow`	
a `</>` *b*	Slash left-right arrow	`slashLeftRightArrow`	
a `-/-` *b*	Slash line	`slashLine`	
a `==` *b*	Equality	`eq`	
a `!=` *b*	Inequality	`neq`	
a `&` *b*	Bitwise and	`and`	
a `^` *b*	Bitwise exclusive or	`xor`	
a `|` *b*	Bitwise inclusive or	`or`	
a `&&` *b*	Conditional and	`cand`	
a `||` *b*	Conditional or	`cor`	
a `::` *b*	Guard	`guard`	Sect. 6.3.4
a `?` *b* `:` *c*	Conditional	-	
a `:` *b*	Range	`range`	Sect. 6.3.2
a `=` *b*	Assignment	-	
a `:=` *b*	Deferred assignment	`defAssign`	Sect. 6.10
a *op*`=` *b*	Compound assignment	*s*`Assign`	
a `:`*op*`=` *b*	Compound deferred assignment	`def`*S*`Assign`	Sect. 6.10
	op $\in$ {`**`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `^`, `|`}	*s* $\in$ {`pow`, `mul`, `div`, ...}	
		S $\in$ {`Pow`, `Mul`, `Div`, ...}	

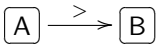
7

Pattern Implementation and Matching Algorithm

The run-time implementation of the semantics of a query requires a pattern-matching algorithm which finds all matches of the query. This algorithm is a fixed component of the run-time system of the XL programming language. Following the divide-and-conquer paradigm, it is spread over several classes in the package `de.grogra.xl.query` which are responsible for specific types of patterns (e. g., node patterns, edge patterns, transitive closures, compound patterns), see the lower part of the class diagram in Fig. 7.1 on the next page. The whole pattern graph of a query is then represented as a hierarchy of such patterns.

7.1 Common Semantics of Patterns

Each pattern has to find valid matches for the query variables with which it is linked. Some query variables may already be bound when a pattern is asked to find matches, then the already bound variables have to be taken as constraints, and only the remaining query variables have to be matched. For example, the simple pattern graph



consists of two type patterns for nodes (A and B) and an edge pattern which matches successor edges. The edge pattern shares its query variables for the source and target node with the incident node patterns. If the pattern graph is matched from left to right, at first the A pattern has to scan the whole graph for nodes of type A. It binds them to a query variable which also represents the source node of the edge pattern. The edge pattern takes this query variable as fixed input, finds all outgoing successor edges of the source node and binds the edge and its target node to corresponding query variables. Finally, the B pattern has to check whether the already bound value of its single query variable has type B.

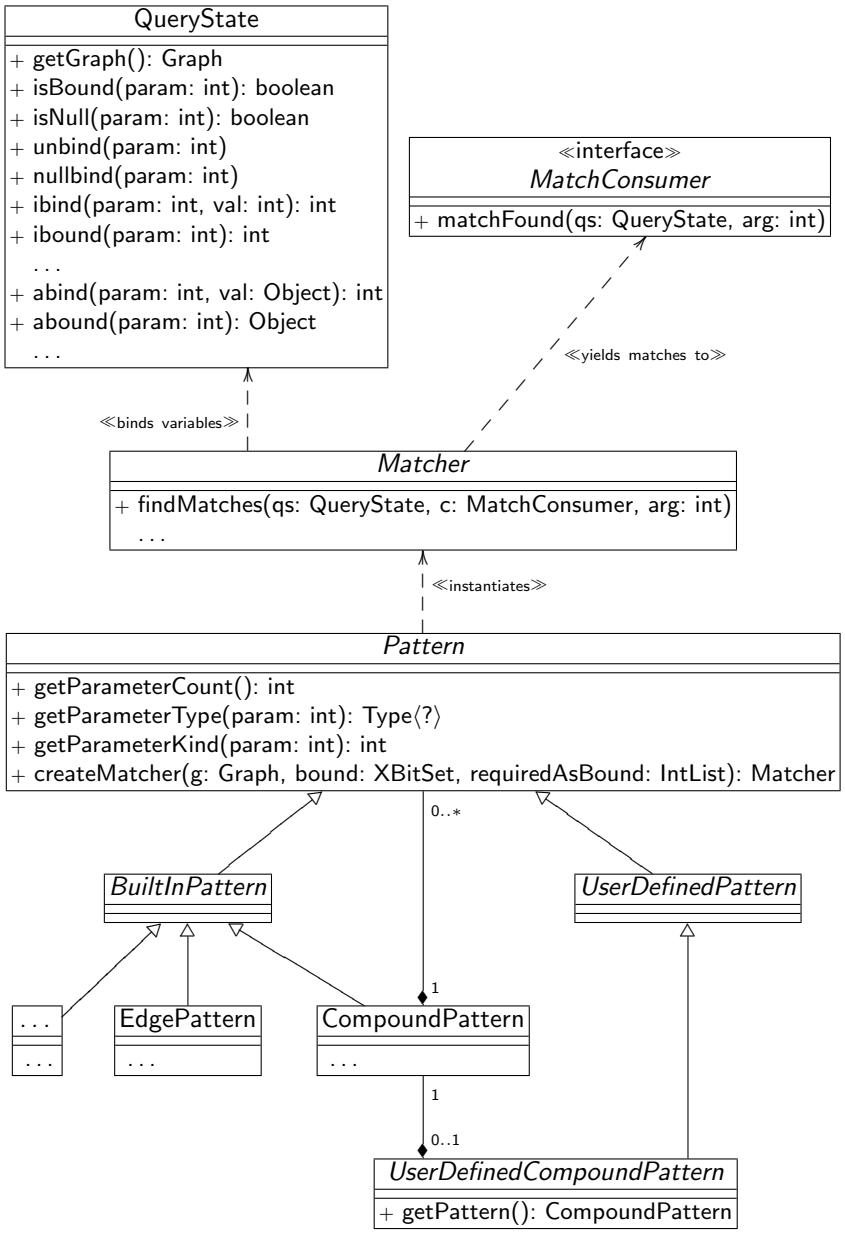


Figure 7.1. Class diagram for patterns and their matching

A pattern is linked with query variables by its *parameters*. For example, a simple node pattern A has a single parameter of type A which is associated with a query variable of the query. An edge pattern has three parameters, where the parameters for the source and target node are associated with the same query variable as the parameters of the incident node patterns (see also the discussion how patterns are combined in Sect. 6.5.11 on page 156). In general, the number n of parameters of a pattern is given by the method `getParameterCount`. For each parameter indexed from 0 to $n-1$, the method `getParameterType` determines the type of the parameter, and `getParameterKind` the kind, which is a combination of the bit masks `NODE_MASK`, `INPUT_MASK`, `OUTPUT_MASK`, `CONTEXT_MASK` with the following meanings:

- If `NODE_MASK` is present, the parameter is a *node parameter* whose associated query variable contains nodes. Such a variable is treated specially when injective matching is required: then two different variables for nodes must not contain the same value.
- The mask `INPUT_MASK` indicates an *input parameter*. Its variable has to be bound to a value before the pattern is asked to find matches.
- The mask `OUTPUT_MASK` indicates an *output parameter*. This is just a hint to the search plan generator (Sect. 7.3.1 on page 197) which indicates that it is a very cheap operation for the pattern to bind values to the associated variables.
- If `CONTEXT_MASK` is present, the query variable is considered to be in the context of the query. Context variables are treated specially if the query is the left-hand side of a rule, see Sect. 7.6 on page 201.

The most important method of `Pattern` is `createMatcher`: this method creates a `Matcher` which is able to find matches for the specific pattern in a `Graph`. The parameter `bound` of `createMatcher` is a bit set which indicates the parameters of the pattern whose query variables are already bound when the matcher is invoked (i. e., the set contains bit j if and only if the query variable of parameter j is already bound when `findMatches` is invoked on the matcher). If the matcher needs additional parameters being bound in advance, `createMatcher` has to add their indices to the list `requiredAsBound`. If for the initial configuration `bound` no matcher can be created, `createMatcher` has to return **null**.

The single abstract method `findMatches` of the class `Matcher` receives a current `QueryState`, a `MatchConsumer` and some argument `arg`. Within the process of matching, the binding of query variables is performed through the methods of the query state: `isBound` tells whether the query variable of the parameter with index `param` is currently bound to a value. If so, the methods `ibound`, `lbound`, `fbound`, `dbound`, `abound` can be invoked to determine the value, where the method has to correspond to the type: values of type **boolean**, **byte**, **short**, **char** and **int** are obtained through `ibound`, where non-**int** values are encoded as **int** values in the same way as for the Java virtual machine, the

other methods are responsible for the types **long**, **float**, **double** and `Object`, respectively. If some query variable has not yet been bound, the matcher has to do this via one of the `bind` methods. Their return value indicates the success of binding: the value `BINDING_PERFORMED` represents a successful binding, the value `BINDING_MATCHED` means that the variable has already been bound to the same value, and `BINDING_MISMATCHED` means that the variable has already been bound to a different value so that the desired binding does not match the existing one. When all query variables have been bound, the matcher has found a match for its pattern, and the match consumer has to be invoked with the query state and `arg` as arguments. Afterwards, the method `unbind` of the query state has to be invoked to remove the binding for newly bound query variables.

If optional patterns (Sect. 6.5.7 on page 153) are used, the value of a query variable may be a null value which indicates that no match for the optional part could be found. Whether this is the case is determined by the method `isNull` of the query state, and a null value can be bound to a query variable by the method `nullbind`. This is different from binding the value **null** via `abind`: in the latter case, **null** is regarded as an actual value, and `isNull` returns **false**.

7.2 Built-In Patterns

The implementation of most of the built-in patterns is straightforward. For example, an edge pattern expects at least one of the query variables for its incident nodes to be bound in advance. The matcher of an edge pattern then delegates to the method `enumerateEdges` of the current `Graph` (see Fig. 6.1 on page 143), i. e., which edges a given node has is determined by the graph model.

The `TransitiveClosure` pattern makes use of a set of package-visible methods of the query state to create a new set of query variables for each recursion depth. This is necessary in order to keep track of the complete binding of query variables for each currently instantiated repetition of the contained pattern.

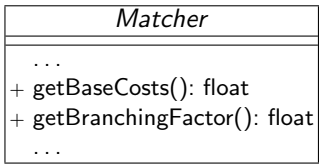
7.3 Compound Pattern

The implementation of the matcher for compound patterns is crucial with respect to the time efficiency of pattern matching. The matcher has to decide in which order subpatterns are matched, and this has a great influence on the required computation time. An extreme example is a complex pattern which contains a subpattern for nodes of a type of which there are no instances in the current graph. If this subpattern is matched at first, the algorithm will

terminate quickly. On the other hand, if it is matched at last, a lot of computation time will be wasted in finding matches for the other subpatterns, which then cannot be extended to the whole pattern. In general, wrong matching attempts should be terminated as early as possible according to the first-fail principle in constraint satisfaction systems [200].

7.3.1 Search Plans and Their Cost Model

The order in which subpatterns of a compound pattern are matched is called a *search plan* in the literature [200, 80, 7]. Finding good search plans requires a cost model in order to compare different search plans and to determine which search plan is good or even optimal. Our cost model is similar to the ones in [200, 7]: for each matcher m of a subpattern, there is an estimated *base cost* $c(m)$ for a single invocation of the matcher and a *branching factor* $f(m)$ which is the estimated number of matches per invocation. These values are obtained by methods of the matcher:



Now the estimated total costs of a search plan of n matchers $m_1, \ldots, m_n$ are given by the formula

$$C(m_1, \ldots, m_n) = \sum_{i=1}^{n} c(m_i) \prod_{j=1}^{i-1} f(m_j) \; . \tag{7.1}$$

I. e., the base costs for each matcher are multiplied with the estimated number of its invocations, and the sum of these products gives the total costs. The cost model of [7] is a special case of (7.1) if we set $c(m) = f(m)$.

Currently, the values for base costs and branching factors of patterns are chosen heuristically without reference to the current graph. This could be improved by an analysis of the actual costs and branching factors for typical graphs. Ideally, the values would be chosen on the basis of the current graph. For example, the exact branching factor of a pattern for nodes of some given type is the current number of such nodes in the graph. This number could be provided by the graph as part of some graph statistics. Of course, it may vary from one evaluation of a query to the next.

7.3.2 Generating a Search Plan

The current implementation of compound patterns computes the optimal search plan by a recursive backtracking algorithm. In order to not explore

the full space of search plans, the algorithm sorts the not yet included matchers with respect to their branching factor and appends matchers to the search plan in increasing order of branching factor. If the total cost of the currently constructed (partial) search plan is greater than the total cost of the currently optimal search plan, the current recursion branch is terminated since it cannot lead to a new optimal search plan.

The whole algorithm is invoked once per query, namely when the query is evaluated for the first time. Thus, the computation time required for the generation of an optimal search plan accrues only once and can be neglected for practical applications.

Of course, the best solution with respect to computation time of the query itself (i. e., without the time for search plan generation) would be to compute an optimal search plan for each evaluation of the query based on the current graph. But then, the computation time for search plan generation dominates in some situations. This could be an interesting starting point for future work: is there an efficient way to detect when it pays off to generate a new search plan, and if so, how can this be implemented? One could observe changes in the statistics of the graph (e. g., number of nodes or edges of specific types). Most likely, then also the algorithm to find an optimal search plan has to be improved with respect to computation time. This might result in an algorithm which does not find the optimal search plan, but a good one in reasonable time.

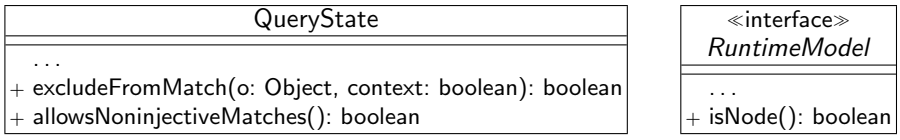
Patterns of the class `UserDefinedCompoundPattern` are treated specially when a search plan is generated. These patterns wrap a `CompoundPattern`. Now the usual way to generate a search plan would be to treat the wrapped compound pattern as an atomic pattern within the search plan. However, if we flatten the hierarchy and replace the compound pattern by its components in the search plan, it might be possible to find a better search plan. Thus, for performance reasons we leave the divide-and-conquer paradigm and replace a compound pattern within a compound pattern by its components when a search plan is generated. This approach is also mentioned in [194].

7.3.3 Enumeration of Nodes

Patterns typically require some of their query variables for nodes to be already bound when their matcher is invoked. For example, an edge pattern requires either the source node or the target node to be bound in advance. If such a pattern is used within a compound pattern and no other subpattern has bound a node to the query variable, compound patterns add an implicit pattern of class `de.grogra.xl.query.EnumerateNodesPattern`. The matcher of the latter pattern obtains all nodes which are instances of the type of the query variable by invocation of the method `enumerateNodes` on the current graph (Fig. 6.1 on page 143, this might be a costly operation with a large branching factor).

7.3.4 Checking Constraints

After each found match for a subpattern, the matcher for compound patterns checks constraints for node parameters. The constraints are given by three methods in the types `QueryState` and `RuntimeModel`:



For each node parameter of a subpattern, the bound value has to be a node as defined by the method `isNode` of the current run-time model. Furthermore, the method `excludeFromMatch` is invoked on the query state with the node as first argument and a **boolean** value as second argument which is **true** if and only if the node parameter is in the context of a query or not a direct part of the query of a rule. If the result of the invocation is **true**, the current match for the subpattern is rejected. Finally, if the current (partial) match for the whole query already contains a binding of the same node to another query variable, the invocation of `allowsNoninjectiveMatches` on the query state returns **false**, and if there is no folding clause (Sect. 6.5.9 on page 155) which allows the binding of both query variables to the same node, the match is rejected, too, in order to ensure injectivity of the match with respect to nodes. But note that the match may be noninjective with respect to non-nodes.

Using the method `excludeFromMatch`, a useful mechanism known from L-system software like GROGRA and L-Studio (Sect. 3.15.1 on page 35 and Sect. 3.15.2 on page 36) can be implemented. This software handles the case that there are several rules with the same symbol as predecessor in a different way than the theory: while an L-system with such rules is nondeterministic in theory according to Def. 3.1 on page 18, in practice the first rule (in textual order) is chosen for which the application conditions are fulfilled. The application conditions may be the context, some conditions on the parameter values of the symbol, or a stochastic condition to implement stochastic L-systems. Within the framework of relational growth grammars, we can generalize this mechanism. Rules are executed in order of the control flow of the XL programming language, and each execution of a rule is able to detect which nodes will be deleted on derivation as a result of the execution. These nodes have been "consumed" in a way and should not be used for further matches of the same parallel derivation. So each node which is known to be deleted on derivation is excluded from the set of candidates for further matches. Note that this is not a requirement specified by the XL programming language, but just a possible application of the mechanism of matchers of compound patterns if the method `excludeFromMatch` is implemented suitably by a subclass of `QueryState`. This is done by the XL base implementation, see Sect. 9.1.5 on page 245.

7.4 User-Defined Patterns

As it has been explained in Sect. 6.5.2 on page 145 and Sect. 6.5.11 on page 156, user-defined patterns are subclasses of `UserDefinedPattern` which declare a special `signature` method to define the parameters. They have to implement the abstract method `createMatcher` and return a suitable matcher. As an example, the complete implementation of the class `X` and its parameterized pattern from Sect. 6.5.2 on page 145 could be the following:

```
class X extends Node {
    float attr;

    class Pattern extends UserDefinedPattern {
        private static void signature(@In @Out X node, float attr) {}

        public Matcher createMatcher(Graph graph, XBitSet bound,
                                     IntList requiredAsBound) {
            if (!bound.get(0))
                // we need the node parameter as input for the matcher
                requiredAsBound.add(0);
            return new Matcher(1) {
                public void findMatches(QueryState qs, MatchConsumer c,
                                        int arg) {
                    Object o = qs.abound(0); // obtain node
                    if (!(o instanceof X))
                        return; // not an instance of X, does not match
                    // now bind value of attr to variable 1
                    switch (qs.fbind(1, ((X) o).attr)) {
                        case QueryState.BINDING_PERFORMED:
                            // OK, value has been bound
                            c.matchFound(qs, arg);
                            // remove binding of value
                            qs.unbind(1);
                            break;
                        case QueryState.BINDING_MATCHED:
                            // OK, but the same value has already been
                            // bound to the variable
                            c.matchFound(qs, arg);
                            break;
                    }
                }
            };
        }

        public int getParameterKind(int param) {
            return (param == 0) ? INPUT_MASK : OUTPUT_MASK;
        }
    }
}
```

The case that the result of the binding is `BINDING_MATCHED`, i. e., that the query variable for `attr` has already been bound to the same value as `o.attr`, can for example occur for patterns like `n:X(7)` where all `X` nodes with a fixed `attr` value of 7 shall be found.

7.5 Storage of Named Query Variables

If query variables are named by an identifier like the query variables for the node and a parameter in `x:X(a)` (the identifiers being `x` and `a`, respectively), they can be used by the programmer like any other (final) local variable. For example, they may be used in expressions within the query itself as in `x:X(a), (a > 1)`, or they may be used outside of the query, but within their scope as in

```
for ((* a:A b:B *)) {
    System.out.println(a + " has successor " + b);
}
```

However, the internal representation of query variables has to be very different from that of local variables. Local variables are stored as part of a frame within the stack of the Java virtual machine. This cannot be used for query variables as the latter have to be accessible by the query state, but there are no methods to access the stack of the Java virtual machine. Thus, query variables have to be stored at another place where they can be addressed by method invocations. For this purpose, an instance of the interface `de.grogra.xl.query.Frame` is used which provides storage for a set of query variables. The latter are represented by instances of `de.grogra.xl.query.Variable` which the query state uses to obtain and modify their values within its frame, see Fig. 7.2 on the following page. A compiler for the XL programming language has to choose a suitable implementation of these interfaces, and it has to produce code which provides a frame and its variables to the query state (see also Sect. 8.4.1 on page 213).

7.6 Support for Application of Rules

If a query is the left-hand side of a rule, we need an additional functionality of the pattern implementation, namely the possibility to delete (parts of) matches. In principle, the decision when and which objects shall be added or deleted by the application of a rule has to be made by the producer which is used for the right-hand side (see Sect. 6.8.2 on page 173), but in order to be able to do so, the producer needs some support by the query implementation:

1. The producer has to know the nodes and edges of the match.
2. Deletion of objects which are in the context of the left-hand side (Sect. 6.5.8 on page 154) has to be prevented.

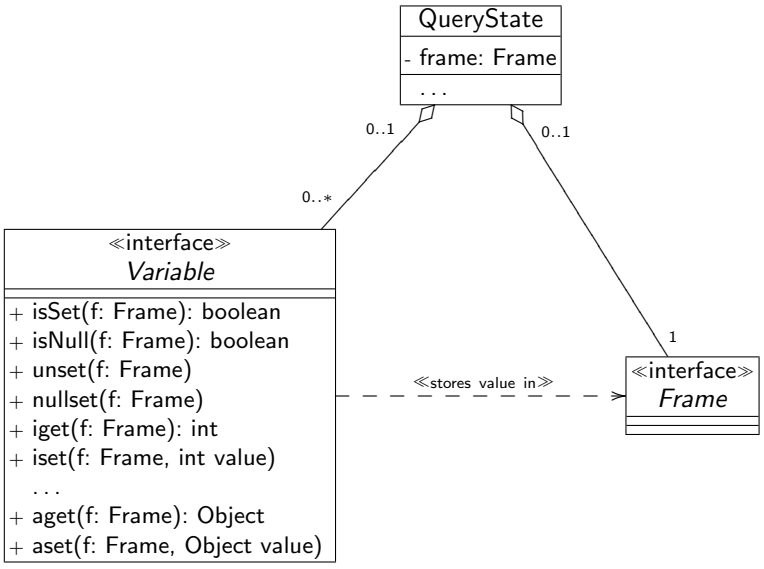


Figure 7.2. Class diagram of frames for query variables

3. For a user-friendly integration of rules of L-system type, the producer has to be able to add implicit connection transformations according to Def. 5.6 on page 100. For this, it needs knowledge about the matches of the textually leftmost and rightmost node patterns. E. g., for a rule `A B ==> C D;` the producer should implicitly add connection transformations from the matched `A`-node to the new `C`-node and from the matched `B`-node to the new `D`-node.

The methods shown in Fig. 7.3 on the next page provide this support. The query state maintains a map from matched nodes to instances of `NodeData`. These instances are also available as a linked list, the anchor being given by the method `getFirstNodeData` of the query state. This addresses the first issue with respect to nodes. The flag `context` of a node data indicates whether the matched node is within the context, this addresses the second issue with respect to nodes. For the third issue, the methods `getInValue` and `getOutValue` return the matches of the textually leftmost and rightmost node patterns, respectively (to be more precise, of the in- and out-parameters of the whole compound pattern of the query, see Sect. 6.5.11 on page 156). Finally, the invocation of the method `visitMatch` on the query state is delegated to the whole pattern hierarchy by invoking `visitMatch` on each matcher whose pattern is not part of the context. The implementation of this method for the matcher of edge patterns invokes `producer$visitEdge` on the producer, this addresses the first two issues with respect to edges.

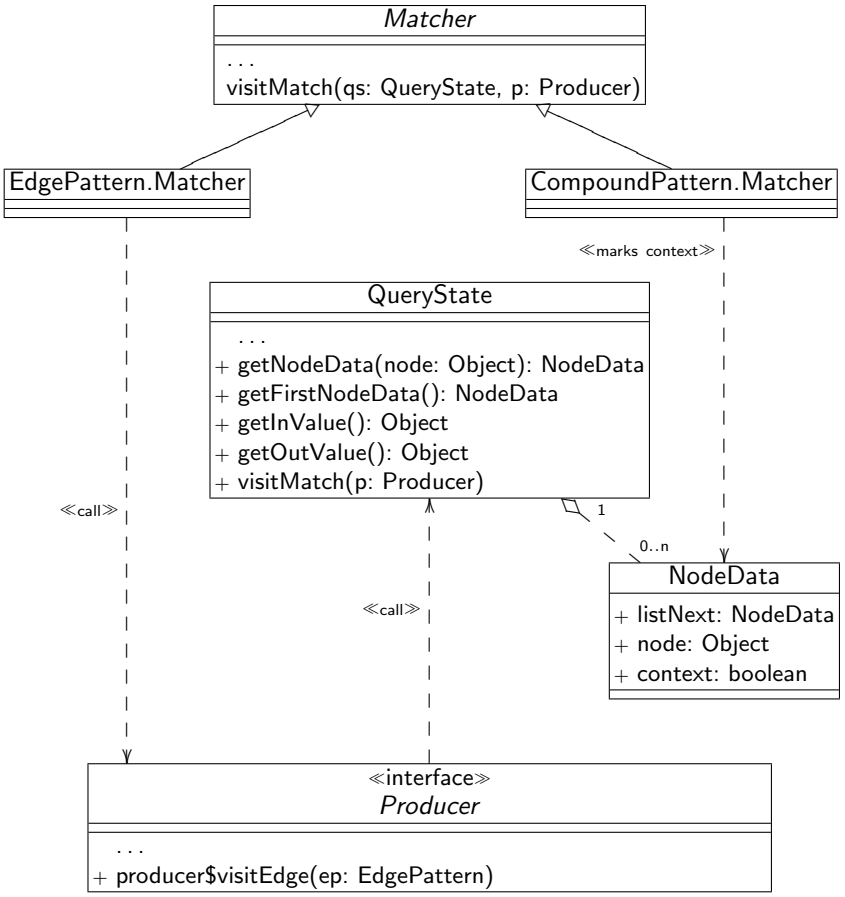


Figure 7.3. Class diagram of support for application of matches

8

Compiler Implementation

In order to execute a program written in the XL programming language, a *compiler* is needed. In general, a compiler translates a program written in a source language into an equivalent program in a target language. A set of highly developed techniques assist in constructing a compiler, having their basis in language theory, but also in algorithms and data structures. The famous "Dragon Book" [2] is *the* standard work in the field of compiler construction, it was used for the implementation of our compiler. The terminology of the book will be used in the sequel.

In our special case, we may think of several source languages: obviously, the XL programming language has to be one of the supported source languages of the compiler. But, within the context of the GroIMP software, it is also desirable to support the language for sensitive growth grammars of the GROGRA software and, if necessary, other existing L-system based systems. As we will see later in Sect. B.15 on page 414, such additional source languages can be integrated if there exists a semantics-preserving mapping from the syntax of the source language in question to the syntax of the XL programming language.

Usually, the target language of a compiler is the machine language of a computer. This computer may be virtual – a *virtual machine*. Several virtual machines have been defined in the past, among them the P-Code machine [135], the Java virtual machine (JVM for short, [120]), and the virtual machine of the Common Language Runtime (CLR for short, [42]). Although these machines are equipped with a sufficiently large instruction set and runtime system such that it should be possible to compile from a large variety of source languages, they were designed with a specific source language in mind: UCSD Pascal in case of the P-Code machine, Java for the Java virtual machine, and several .NET-languages like C# for CLR. The compilation from these source languages to the language of their virtual machine is then relatively straightforward, while other source languages may require intricate constructions.

Since the XL programming language is specified as an extension of the Java programming language, we chose the language of the Java virtual machine as target language of the compiler. As we will see in Sect. 8.5 on page 224, there is a translation of every feature of the XL programming language to *bytecode* (code for the JVM), although some features require additional runtime support and have no direct representation as a short and simple sequence of bytecode.

According to [2], a compiler can be logically organized into consecutive *phases*, see Fig. 8.1: the input stream of characters is grouped into a token stream by the lexical analyzer, this token stream is structured hierarchically by the syntax analyzer according to the syntax of the source language. The result is an (abstract) syntax tree and processed by the semantic analyzer. Its output is translated to intermediate code. These first phases are known as the *front end* of the compiler, the intermediate code being the interface to the *back end* which optimizes intermediate code and, in a final step, generates code for the target language.

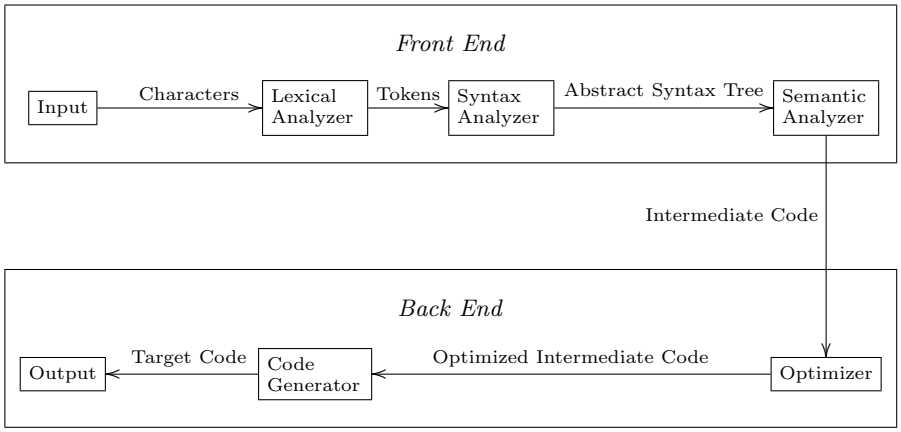


Figure 8.1. Structure of a compiler

The code for virtual machines shares characteristics with intermediate code. For example, the JVM and CLR are stack machines, code for which is a popular form of intermediate code. Generating code for such a virtual machine considerably simplifies the task of writing a compiler: basically, one only has to write the front end of the compiler, the back end being trivial because the intermediate code is the final code. We can also assign the role of the back end to the implementation of the virtual machine: then a compiler for a virtual machine is actually just the first half, namely a front end with a clearly defined interface (the virtual machine) to the back end. It is the task of the implementation of the virtual machine to perform optimization and code

generation for the concrete machine and, in doing so, to complete the compiler pipeline from source code to target code. For example, Sun's HotSpot virtual machine reads bytecode for the JVM, performs optimizations on the code and produces true code for the target machine, but only for code being considered a hotspot (executed often, which is a dynamic property). Its optimizations include but are not limited to inlining, constant propagation and folding, global value numbering, loop unrolling, and range check eliminations.

Given the previous remarks, the structure of the XL compiler is summarized by Fig. 8.2. The individual phases will be described in the following sections. As implementation language, Java was chosen; the figure also shows the names of the implementing Java classes. It turned out that an intermediate code representation in terms of an *expression tree* is advantageous, this will be discussed in Sect. 8.3 on page 209.

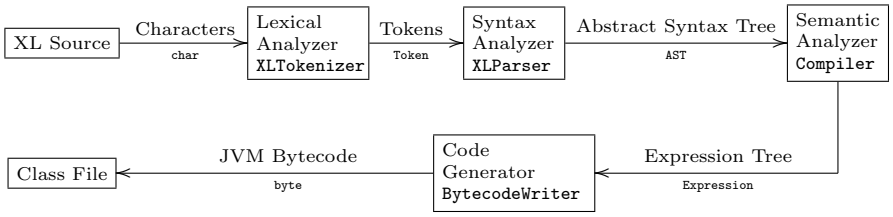


Figure 8.2. Structure of the XL compiler

8.1 Lexical Analysis

During lexical analysis, the character stream representing the program in the source language is grouped into a stream of tokens (terminal symbols of the source language). There are several techniques for the construction of *scanners* which perform this task: At first, they can be coded by hand. This is most flexible, since it does not pose restrictions on the lexical structure of the source language. If the lexical structure can be described by regular expressions, there exist tools like `lex` to automatically create a scanner having a deterministic finite automaton as its basis. Similar tools are available if the lexical structure can be described by a context-free grammar, which is more general than regular expressions.

We chose to implement the scanner by hand, represented by the classes `de.grogra.xl.parser.XLTokenizer`. The advantage is that it can be reused for a set of source languages: instead of encoding the set of keywords and operators indirectly by the transition diagram of a finite automaton, they are represented directly in tables. By filling these tables with different sets of

keywords and operators, the scanner is used not only for the XL programming language, but also the Java programming language, the language of sensitive growth grammars of the GROGRA software, and some pure data formats as part of the GroIMP software.

The scanner implements the interface `antlr.TokenStream` so that it can be used as input to ANTLR generated parsers [140]. This interface consists of the single method

```
antlr.Token nextToken() throws antlr.TokenStreamException;
```

which simply returns the next token in the stream.

8.2 Syntax Analysis

During syntax analysis, the incoming token stream is structured hierarchically according to the syntax by the *parser*. Again, if the syntax can be described by a context-free grammar, there exist tools to create a parser from a grammar definition. A popular parser generator is `yacc` or its GNU implementation `bison`, however, both generate C source code. The tool ANTLR [140] generates Java code, thus it was chosen to generate the parser for the XL programming language.

While `yacc` produces *LR*-parsers, ANTLR produces linear approximate $LL(k)$ parsers [141]. The addition *linear approximate* refers to the lookahead. A full $LL(k)$ parser uses the sets $\mathrm{FIRST}_k(\alpha)$ (the set of all strings up to length k that can begin any sentence derived from the sentential form α) and $\mathrm{FOLLOW}_k(A)$ (the set of all strings up to length k that can appear immediately to the right of nonterminal A in some sentential form) in combination with the current lookahead in order to disambiguate between alternatives. ANTLR's linear approximation simplifies these sets by merging the possible terminals at a given lookahead depth. E.g., as FIRST set it considers all strings $x_1 \dots x_j, j <= k$ such that for every terminal x_i there is a string in $\mathrm{FIRST}_k(\alpha)$ which has x_i as its i-th symbol, and there exists some string of up to j terminals in $\mathrm{FIRST}_k(\alpha)$. The benefit of the approach is the reduced lookahead complexity (from $\mathcal{O}(|T|^k)$ to $\mathcal{O}(k|T|)$ where $|T|$ is the size of the input vocabulary), but of course, there are situations where this leads to an artificial nondeterminism and which must therefore be handled specially by additional mechanisms.

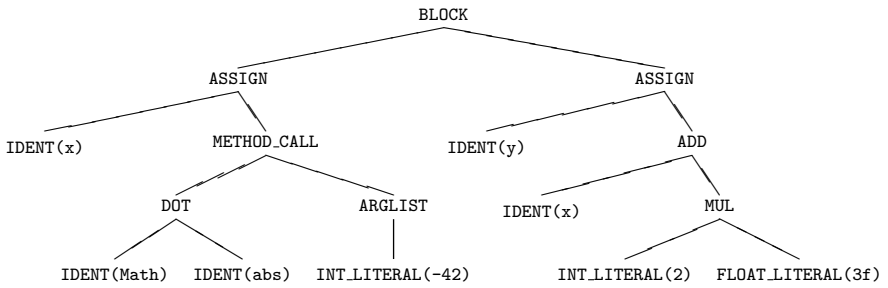
The parser for the XL programming language uses a linear approximate lookahead of $k = 2$ tokens together with *syntactic predicates* wherever this finite lookahead is not sufficient to disambiguate alternatives. Syntactic predicates are a feature of ANTLR and make use of backtracking in order to allow an arbitrary lookahead. The output of the parser is an abstract syntax tree whose nodes are instances of the interface `antlr.collections.AST`. This tree is generated in a complete pass, i. e., the action of the parser is not interwoven with the semantic analyzer.

As a simple example, consider the following fragment of input:

```
{ x = Math.abs(-42);
  y = x + 2 * 3f; }
```

Listing 8.1. Source code fragment

This is translated by the parser to the following abstract syntax tree, where lexemes of identifiers and literals are shown in parentheses:



8.3 Semantic Analysis and Expression Tree Generation

The semantic analysis is the main part of the XL compiler. Its tasks are manifold, they include:

- Declared entities have to be stored in symbol tables.
- The meaning of names has to be determined; e. g., whether a name in an expression refers to a local variable, an instance variable, a class variable, or some other entity, or nothing so that a semantic error has to be reported.
- Static type checking has to be performed.
- Overloading of operators and methods has to be resolved on the basis of the argument types.
- Constant expressions have to be evaluated.
- Control flow analysis has to verify that no uninitialized variables are read, that final class and instance variables are definitely assigned exactly once, and that there does not exist unreachable code.
- An intermediate code representation has to be constructed.

8.3.1 Passes of the Semantic Analysis

Semantic analysis is performed in the style of a *tree walker* which visits nodes of an abstract syntax tree and performs some node-specific actions. Three passes are used: The first pass traverses the abstract syntax tree and collects class, interface and field declarations in symbol tables. Likewise, the second pass collects method and constructor declarations. The third pass traverses the abstract syntax tree for the last time and creates an intermediate code representation in terms of an *expression tree*.

While the use of more than one pass obviously leads to an increase of compilation time, this is a common technique to deal with forward references. The Java and XL programming languages generally allow forward references, the potential places of forward references are:

- Types (classes, interfaces and modules) may be forwardly referenced by **extends** and **implements** clauses of type declarations, by the type of field declarations, by parameter and return types and **throws** clauses of method declarations (we subsume constructors under methods), by annotations, and within expressions.
- Fields may be forwardly referenced within expressions and by module declarations with inherited fields (Sect. 6.11 on page 179). Note that the latter kind of forward reference is needed to declare the constructor of the module class and the signature method of the pattern class of the module, and to implement the constructor body and the pattern class.
- Methods may be forwardly referenced within expressions.
- There are no forward references to entities declared within statements and expressions (local variables, local classes, labels).

The dependence of expressions on types, fields and methods is resolved by the split between the first two passes (declaration of types, fields and methods) and the third pass (compilation of statements and expressions to intermediate code). The dependence of module constructors and signature methods on types and fields is resolved by the split between the first and second pass. Obviously, the dependence of types on types cannot be resolved this way. Instead we have to use placeholders for not yet resolvable references to types and resolve them after the first pass. This is also used for the types of field declarations, so that both type and field declarations can be handled in the first pass.

In principle, the technique of using placeholders (also called *backpatching*) could be used to further reduce the number of passes. But we found that this would unnecessarily complicate the implementation of the compiler with just a slight gain of compilation time. A further disadvantage of multiple passes is the demand on memory, because the complete generated information of one pass has to be kept in memory for the following pass. But again, we found that this disadvantage is justifiable, given a typical XL program and a typical computer on which the compiler runs.

8.3.2 Scopes and their Symbol Tables

The Java and XL programming languages define the notion of a *scope*. Scopes are contiguous regions of a program within which declared entities can be referred to using their simple name. For example, a type (class or interface) declaration induces a scope within which the declared and inherited members of the type are referable by their simple name. The body of a method induces a scope within which its parameters can be accessed. Scopes are nested: e. g.,

the scope of a method body is nested within the scope of its enclosing type declaration.

Scopes are closely related to *symbol tables*, which maintain the association between names and their meaning. Several more or less efficient data structures for symbol tables exist [2], some of which already include the scoping information in the table. For the sake of clean and simple code, we chose a solution where there exists a symbol table for each scope. Namely, we represent scopes by subclasses of `de.grogra.xl.compiler.scope.Scope`, each of which has its own class-specific symbol table. E.g., a `BlockScope` for blocks of statements manages its declared local variables and classes within a symbol table. A `TypeScope` needs to keep track of its declared members, these members are actually stored in an instance of `de.grogra.xl.compiler.CClass` to which the `TypeScope` refers: `CClass` has similar methods to `java.lang.Class`, however, the latter represents a compiled class having been loaded by the JVM, whereas the former represents the currently available information about the class being compiled.

Each scope has a reference to its enclosing scope, this reflects the lexical nesting of scopes and also helps in determining the meaning of a name, which starts at the current scope and progresses to enclosing scopes until a suitable entity of the given name has been found, following the rules of the XL programming language.

8.3.3 Generation of Expression Trees

We chose to not use bytecode, but expression trees as intermediate representation. An expression tree is similar in structure to an abstract syntax tree, but it contains the complete semantic information. For example, the meaning of names and types of operators have been resolved in an expression tree. In principle, it is possible to reuse the abstract syntax tree for the representation of this information by the addition of attributes to its nodes (this is called *annotating* the tree). However, in doing so one frequently has to collect a sequence of basic operations in a single node (e. g., type conversions, implicit **this**). In order to ensure an as direct as possible correspondence between nodes of an expression tree and basic (bytecode) operations, we decided to create an own tree for the representation of expressions (and statements) and their semantic information.

As an example, consider Fig. 8.3 on the next page. It corresponds to the code fragment 8.1 on page 209, where the context scope of the fragment is assumed to contain a local variable `x` of type `java.lang.Integer` and to be part of a class `Test` having a non-static field `y` of type **double**. Compared to its abstract syntax tree, one can identify several differences:

- Token types are replaced by expression classes which represent specific operations.
- Types have been resolved. E.g., the `Add` operation knows that it operates on operands of type **float**.

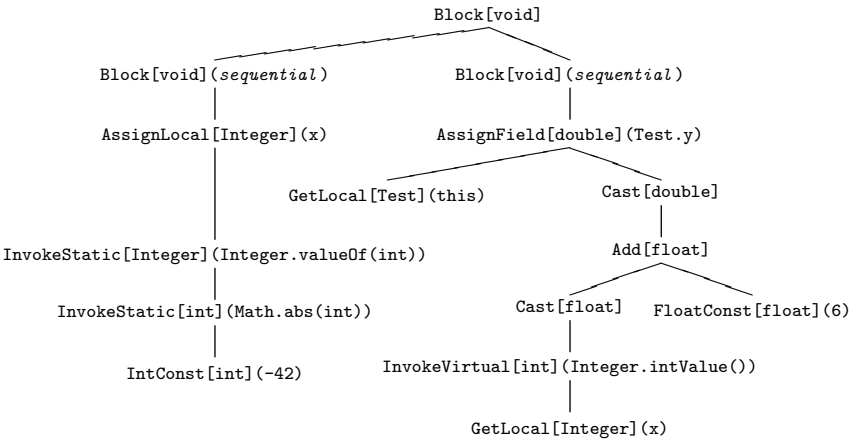


Figure 8.3. Expression tree of example 8.1 on page 209

- Names have been resolved. `Math.abs` has been identified as the static method `abs` of `java.lang.Math` with **int** argument. `x` has been identified as a local variable, so operations `GetLocal` and `AssignLocal` are used to read and assign its value. `y` has been identified as an instance variable, so the operation `AssignField` is used to assign its value. These operations store their associated variable as attribute, so in contrast to the case of the abstract syntax tree, there are no extra nodes for the identifiers of variables.
- Constant expressions have been evaluated.
- There are additional nodes corresponding to casting conversion (`Cast`), boxing conversion (invocation of `Integer.valueOf(int)`), unboxing conversion (invocation of `Integer.intValue()`) and the implicit **this**.
- Expression statements are prefixed by a *sequential* `Block` node.

Besides true expressions like arithmetic expressions or method invocations, also statements (blocks, loops, exception handlers, control transfer statements) are represented within the expression tree, its root being the node for the entire method body. Such statement nodes treat their children as substatements which are executed in the order which is prescribed by the semantics of the statement.

8.4 Extension of the Virtual Machine

In this section, we present some classes and mechanisms which in a way extend the Java virtual machine. They provide support for an analogue of nested routines, namely a stack with the possibility to access enclosing stack frames, and a mechanism to transfer control out of the current method to a specific

point in an enclosing method (informally, we may think of a **goto** to an enclosing method invocation). Although the presented classes do not extend the Java virtual machine in the proper meaning of the word, they do this from a conceptual point of view. In fact, it would be most efficient to actually extend the Java virtual machine to an XL virtual machine by the integration of the functionality of the auxiliary classes as built-in features of the virtual machine. However, this is beyond the scope of the presented work.

The central class of the extension is `de.grogra.xl.vmx.VMXState`. There is a current instance of this class for each thread, it can be obtained by the static method `current`.

The usage of the mechanisms of this section within compiled code of the XL programming language is not mandatory. Our compiler implementation produces code which makes use of the mechanisms, but other compilers may opt to define their own mechanisms or even to produce code for a true XL virtual machine as it has been suggested in the previous paragraph.

8.4.1 Stack Extension

The Java virtual machine provides a stack for each thread. For each invocation of a method, a new frame is pushed onto the stack which contains the local variables for that invocation and an area for the operand stack which holds temporary results. The current frame can be accessed by instructions of the Java virtual machine. There is no possibility to access the current frame by other means, e. g., by access methods, and there is no possibility at all to access other frames than the current frame, even not by instructions of the Java virtual machine.

Being able to access enclosing frames of the current frame would be desirable for the implementation of generator method invocations. Namely, assuming that `produce` is a generator method like the one in Sect. 6.3.1 on page 133, the translation of the code

```
int sum = 0;
for (int value : produce(100)) {
    sum += value;
}
System.out.println(sum);
```

to the conventional Java code

```
int sum = 0;
produce(new IntConsumer() {
        public void consume(int value) {
            sum += value;
        }
    }, 100);
System.out.println(sum);
```

does not work because the `consume` method has no access to the local variable `sum` of the enclosing method. The solution on page 133 wraps the value of `sum` in an array:

```
final int[] sum = {0};
produce(new IntConsumer() {
        public void consume(int value) {
            sum[0] += value;
        }
    }, 100);
System.out.println(sum[0]);
```

This works since the array `sum` itself is declared final and only read within `consume`, so that the value of the local variable `sum` can be copied to an implicitly declared field of the anonymous `IntConsumer` class when the constructor of the latter is invoked. Using arrays can be compared to using pointers: the array `sum` can be regarded as a pointer to its single element.

The drawback of this solution is that it could lead to a lot of heap traffic and garbage if generator methods are invoked frequently: each invocation allocates a new consumer and an array for each local variable which is used within the consumer. Although allocation is a relatively cheap operation for current implementations of the Java virtual machine, the subsequent need for garbage collection is not.

For this reason, we chose to implement a stack similar to the one of the Java virtual machine, but represented as part of the class `VMXState` with methods to access the current and enclosing frames. To be more precise, the stack frame of (direct or transitive) statically enclosing method invocations can be accessed, where the direct statically enclosing method invocation is the nearest invocation of the (textually) enclosing method of the current method. This is discussed in detail in [2] for nested routines which are possible within languages like Pascal, but not for the Java programming language. If we store the variable `sum` on this stack, it is possible to access it within the nested method `consume`.

The stack-related methods of `VMXState` are shown in Fig. 8.4 on the facing page. A new frame is entered by invocation of `enter` on the current `VMXState`, and it is left by `leave`. The current frame can be obtained by invoking `getFrame`. A frame is represented by an instance of `de.grogra.xl.vmx.VMXState.VMXFrame`. It stores a link to its `parent` frame (the frame which has been current on invocation of `enter`) and a `staticLink` to the frame of the statically enclosing method invocation. The get- and set-methods of `VMXState` read and write the value of local variables in the context of the current frame. Local variables are represented by instances of `de.grogra.xl.vmx.VMXState.Local`, where `index` is the storage location within a frame and `nesting` specifies the frame to use. If `nesting` is zero, this is the current frame, otherwise `nesting` static links to frames of statically enclosing invocations have to be followed.

The classes `VMXFrame` and `Local` implement the interfaces `Frame` and `Variable`, respectively, for the pattern matching algorithm (see Sect. 7.5 on page 201). Thus, code generated by our compiler stores query variables within a frame of the `VMXState`.

Besides frame-based access, the stack can also be used by traditional `push`/`pop` methods in the manner of an operand stack. The implementation of `VMXFrame` guarantees that only values added with `push` may be popped with `pop`, i. e., `pop` cannot be used to access contents of a frame.

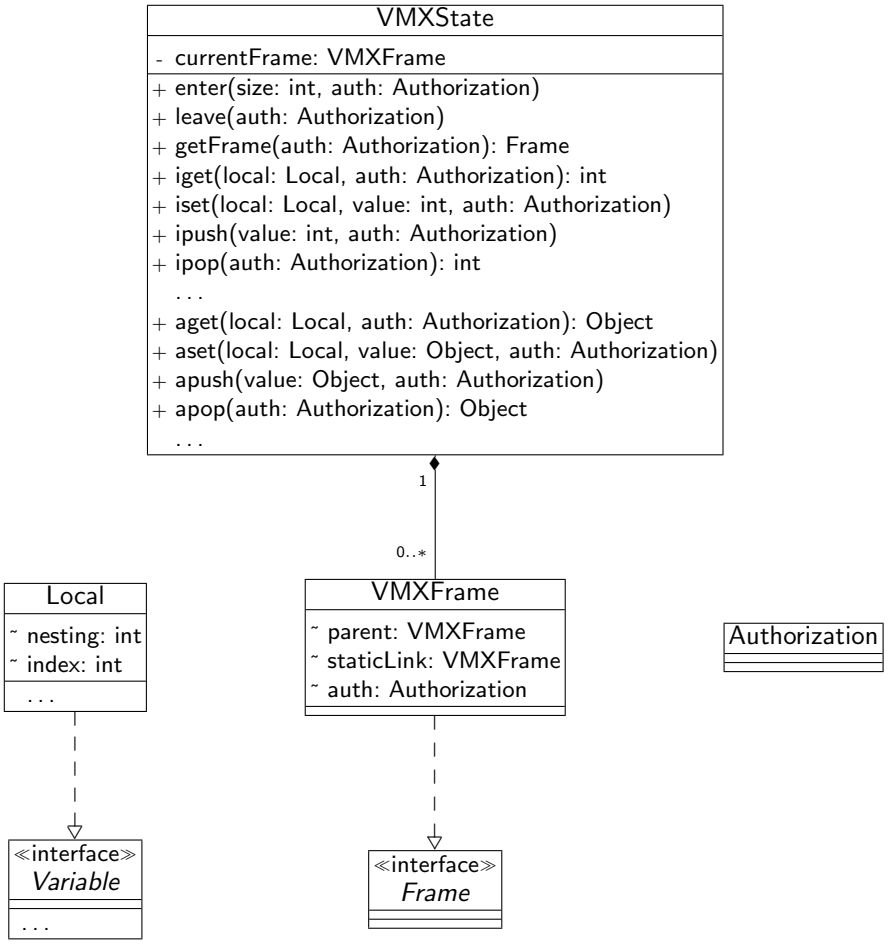


Figure 8.4. Class diagram of the stack extension

Security Considerations

Because the `VMXState` of a thread can be obtained by any code by just invoking `VMXState.current()` within that thread, we have to pay special attention to security issues if we do not want to spoil the sophisticated security mechanism of the Java virtual machine. Namely, if the methods to read and write variables on the stack or to enter or leave frames can be invoked by anyone, malicious code would be able to access variables of enclosing method invocations which might contain sensitive private data. For this reason, each frame is marked with an instance of `de.grogra.xl.vmx.Authorization`, namely with the argument passed to the `auth` parameter of the method `enter`. Each method which accesses a frame has an `auth` parameter, too. Only if this is the same instance as the one of the frame, the access is allowed. Otherwise, a `SecurityException` is thrown. The same holds for the `push`/`pop` mechanism, i. e., the `auth` parameter for `pop` has to match the `auth` parameter of the corresponding invocation of `push`.

The compiler makes use of this mechanism in the following way: for each class which contains a method that needs a frame within the `VMXState`, it declares a private static final field within that class which holds a newly created instance of `Authorization`. Thus, only code within the class itself has access to the instance. This guarantees that the corresponding content of frames may not be accessed by other classes.

Of course, security considerations play no relevant role within applications related to plant modelling. However, as the design of the XL programming language is very general, one may also think of other applications where security is an important issue.

8.4.2 Descriptors for Nested Method Invocations

Using the `VMXState`, our example

```
int sum = 0;
for (int value : produce(100)) {
    sum += value;
}
System.out.println(sum);
```

would be compiled into an equivalent of the conventional Java code

```
private static final Local sumLocal = new Local(0, 0);
private static final Local sumNested = new Local(1, 0);
private static final Authorization auth = new Authorization();
...
final VMXState vmx = VMXState.current();
vmx.enter(1, auth);
vmx.iset(sumLocal, 0, auth);
produce(new IntConsumer() {
        public void consume(int value) {
```

```
            vmx.iset(sumNested, vmx.iget(sumNested, auth) + value, auth);
        }
    }, 100);
System.out.println(vmx.iget(sumLocal, auth));
vmx.leave(auth);
```

What is still missing in this code is a mechanism to set the parent frame of the invocation of `consume` to the frame of the outer code. Languages with support for pointers to nested routines like Pascal solve this by actually using a descriptor instead of a mere pointer [2]. The descriptor contains both a pointer to the routine and a static link to the stack frame which shall become the parent frame of the invocation. This mechanism is implemented in our extension of the virtual machine by a simple callback interface `de.grogra.xl.vmx.Routine` which stands for a pointer to a nested method and a class `de.grogra.xl.vmx.RoutineDescriptor` whose instances store both such a pointer and a stack frame as static link, see Fig. 8.5 on the following page. `RoutineDescriptor` also implements all consumer interfaces by letting the `consume` methods push their received value onto the stack of the `VMXState`. A correct Java code for our example is then

```
private static final Local valueParam = new Local(0, 0);

private static final Routine body = new Routine() {
    // frame for body contains a single value for the "value" parameter
    public int getFrameSize() {return 1;}
    public int getParameterSize() {return 1;}

    public void execute(VMXState vmx) {
        // obtain "value" from frame
        int value = vmx.iget(valueParam, auth);
        vmx.iset(sumNested, vmx.iget(sumNested, auth) + value, auth);
    }
};
...
vmx.iset(sumLocal, 0, auth);
RoutineDescriptor rd = vmx.createDescriptor(body, -1, auth);
produce(rd, 100);
System.out.println(vmx.iget(sumLocal, auth));
```

8.4.3 Control Transfer to Enclosing Method Invocations

The usage of nested methods for generator method invocations not only requires access to enclosing stack frames, but also needs a mechanism to transfer control to an enclosing method. The following code illustrates this:

```
boolean check(Reader chars) {
outerLoop:
    while (true)
```

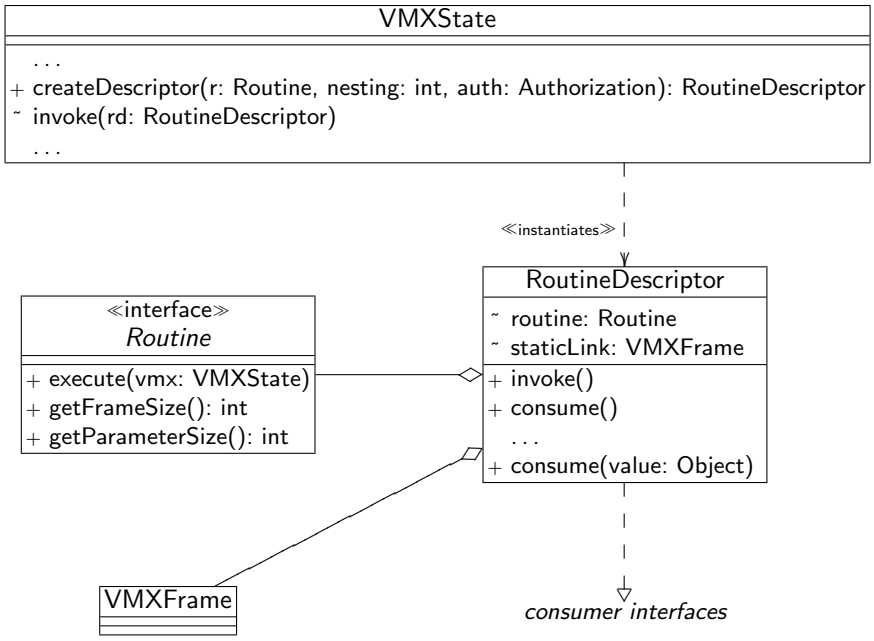


Figure 8.5. Class diagram for routine descriptors

```
    {
        for (char value : prefix()) {
            int i = chars.read();
            if (i < 0) {
                return false;
            } else if (i != value) {
                continue outerLoop;
            }
        }
        return true;
    }
}
```

The method `check` compares the character stream represented by `chars` with the sequence of values yielded by the generator method `prefix`. If the read characters match the sequence, **true** is returned. Otherwise, the test is repeated starting with the first not yet read character. If no match was found, **false** is returned. Now the compiler moves the body of the **for** loop to a nested `consume` method so that

1. the **continue** statement has a target which is in an enclosing method,

2. the **return** statement has to return control to the invoker of the enclosing method,
3. exceptions which might be thrown in the nested body (an `IOException` as a result of `chars.read()` or `NullPointerException` if `chars` is **null**) have to be hidden from the generator method `prefix`. This might not be obvious, but recall that `prefix` invokes the consumer method which contains the body of the **for** loop. So any exception thrown by the body passes through `prefix` and, thus, could be caught there. But the exceptions thrown by the body should not be visible to `prefix` because the latter is completely independent of the body.

The only mechanism which the Java virtual machine provides for such nonlocal transfer of control is that of throwing and catching exceptions. I. e., **continue** and **return** in the above example have to throw an exception which is then caught in the enclosing method and treated accordingly. Also this exception has to be hidden from the intermediate invocation of `prefix`. Unfortunately, a complete hiding is not possible with the the Java virtual machine. The best what we can do is to use a subclass of `java.lang.Error` to transfer control and information from the nested method to the enclosing method as such exceptions are usually not caught by normal code.

The class `de.grogra.xl.vmx.AbruptCompletion` contains subclasses for the different abrupt completions (**break** and **continue**, **return**, **throw**; the term *abrupt completion* is defined in [72] and stands for a transfer of control). Furthermore, it contains the class `Nonlocal` which wraps an abrupt completion and can be thrown to transfer control and information to the enclosing method invocation specified by `frame`. The classes are shown in Fig. 8.6 on the next page.

Exceptions of type `Nonlocal` have to be caught in enclosing method invocations in order to check whether these invocations are the target of the nonlocal transfer of control, i. e., if their frame is the same as the `frame` of the `Nonlocal` instance. This is done by invoking the method `getReason` on the exception. This returns the reason if the frames match, or it re-throws the exception otherwise. The following code shows a Java equivalent of the example:

```
1  private static final Local charsLocal = new Local(0, 0);
2  private static final Local charsNested = new Local(1, 0);
3  private static final Local valueParam = new Local(0, 0);
4  private static final int outerLoopCont = 2;
5  private static final Authorization auth = new Authorization();
6  private static final Routine body = new Routine() {
7      public int getFrameSize() {return 1;}
8      public int getParameterSize() {return 1;}
9
10     public void execute(VMXState vmx) {
11         char value = (char) vmx.iget(valueParam, auth);
12         int i = ((Reader) vmx.aget(charsNested, auth)).read();
```

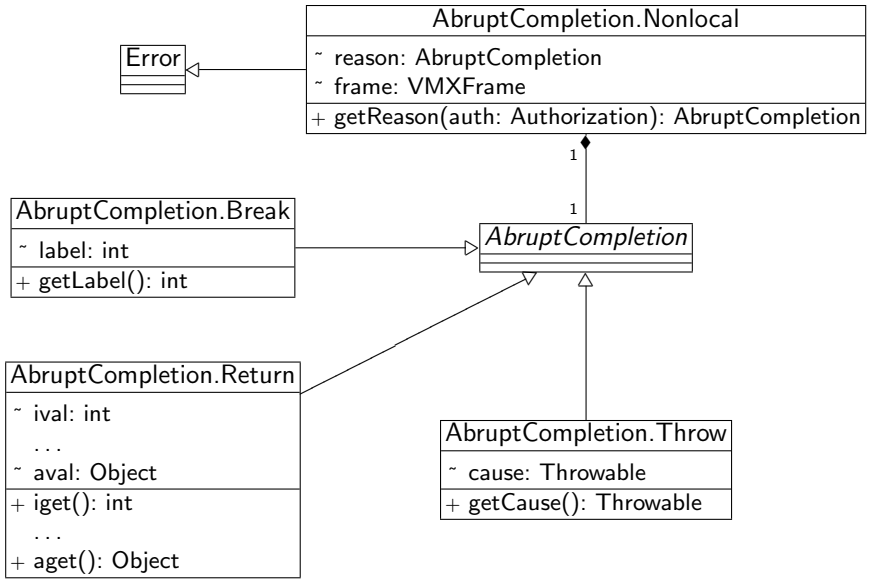


Figure 8.6. Class diagram for abrupt completions

```
13          if (i < 0) {
14              // return false, encoded as int-value 0
15              throw vmx.newNonlocal(1, vmx.ireturn(0), auth);
16          } else if (i != value) {
17              // perform a nonlocal jump to outerLoop
18              throw vmx.newNonlocal(1, vmx.newBreak(outerLoopCont), auth);
19          }
20      }
21  };
22
23  boolean check(Reader chars) {
24      VMXState vmx = VMXState.current();
25      vmx.enter(1, auth);
26      try {
27          vmx.aset(charsLocal, chars, auth);
28      outerLoop:
29          while (true)
30          {
31              RoutineDescriptor rd = vmx.createDescriptor(body, -1, auth);
32              try {
33                  prefix(rd);
34              } catch (Nonlocal e) {
35                  AbruptCompletion reason = e.getReason();
36                  switch (reason.getLabel()) {
```

```
37                      case Throw.LABEL: // reason is a thrown exception
38                          throw ((Throw) reason).getCause();
39                      case Return.LABEL: // decode boolean return value
40                          return ((Return) reason).iget() != 0;
41                      case outerLoopCont:
42                          continue outerLoop;
43                      default: // should not happen for correct code
44                          throw new AssertionError(e);
45                  }
46              }
47          }
48      } finally {
49          vmx.leave(auth);
50      }
51      return true;
52  }
```

If an exception is thrown within the `execute` method (lines 11 to 19), the implemented `RoutineDescriptor`/`VMXState` mechanism automatically wraps this exception in a `Nonlocal` instance with a `Throw` reason. For the other two reasons for a nonlocal transfer of control (the **continue** and **return** statements), the code explicitly creates the wrapping `Nonlocal` instance by invocation of the corresponding factory methods on the current `VMXState` (lines 15 and 18). In any case, the reason for abrupt completion is unwrapped within the enclosing method in line 35 and then handled correspondingly.

8.4.4 Minor Issues

Two further minor issues related to generator method invocations have to be addressed. The first issue arises in situations where a generator method is invoked as part of an expression which already contains some subexpressions to the left of the generator method invocation. Examples are

```
println((x + 1) * produce(100));

for (int i : Math.min(limit(), produce(100))) {
    doSomething(i);
}
```

The subexpressions `x + 1` or `limit()`, respectively, are evaluated before the generator method invocation. However, their values are combined with the yielded values of `produce` within the implicitly generated consumer methods. Thus, these values have to be passed to the consumer methods as auxiliary variables on the stack extension:

```
private static final Local aux0 = new Local(0, 0);
private static final Local aux0Nested = new Local(1, 0);
private static final Local valueParam = new Local(0, 0);
private static final Routine body = new Routine() {
```

```
    public int getFrameSize() {return 1;}
    public int getParameterSize() {return 1;}

    public void execute(VMXState vmx) {
        int aux = vmx.iget(aux0Nested, auth);
        int value = vmx.iget(valueParam, auth);
        println(aux * value);
    }
};
...
vmx.iset(aux0, x + 1, auth);
RoutineDescriptor rd = vmx.createDescriptor(body, -1, auth);
try {
    produce(rd, 100);
} catch (Nonlocal e) {
    ...
}
...
```

The second issue arises when an aggregate method is invoked as part of an expression which already contains some subexpressions to the left, and when the aggregate value is based on a generator method invocation. A simple example is

```
int x = 3 * first(produce(100));
```

The problem here is that the operand stack of the Java virtual machine is not empty when the aggregation starts (for the example it contains a single value 3). The aggregation then invokes the generator method `produce`. But this may result in a `Nonlocal` exception caught within the aggregation, for example if the aggregation wants to terminate `produce` because the aggregated value is already known (see Sect. 6.4.2 on page 138). Now the Java virtual machine clears the whole operand stack on catching an exception, so the still needed stack content would get lost. For this reason, we have to move the exception handler to an auxiliary method `invokeProduce` (which has its own operand stack):

```
static AbruptCompletion invokeProduce(IntConsumer cons, int param) {
    try {
        produce(cons, param);
        return null;
    } catch (Nonlocal e) {
        return e.getReason();
    }
}
...
RoutineDescriptor rd = vmx.createDescriptor(body, -1, auth);
AbruptCompletion reason = invokeProduce(rd, 100);
if (reason != null) {
    switch (reason.getLabel()) {
```

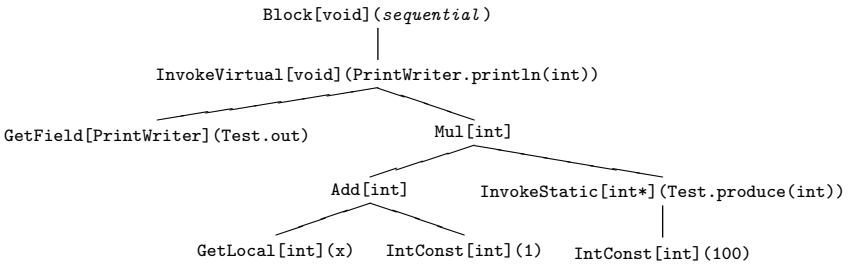
```
        ...
    }
}
...
```

8.4.5 Expression Tree Transformation for Invocations of Generator Methods

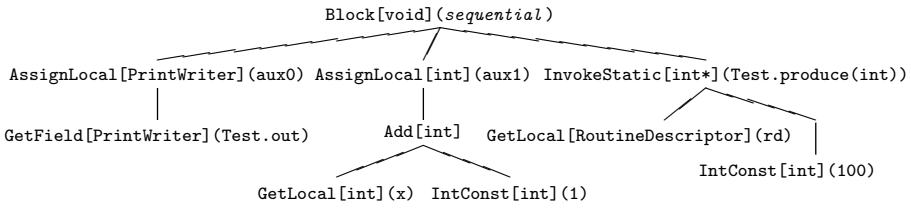
As it has been discussed on the previous pages, the invocation of generator methods has to be handled specially. Although we have shown Java equivalents of the required code for generator method invocations, the compiler performs transformations not at the level of source code, but on expression trees. It constructs a valid expression tree out of a preliminary one with not yet valid invocations. As an example, the code

```
out.println((x + 1) * produce(100));
```

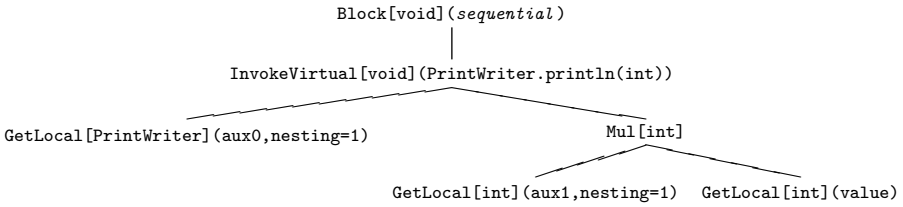
results in a preliminary expression tree



As this contains the generator method invocation of `produce`, a transformation has to be applied. The nearest enclosing `Block` which is marked as *`sequential`* is used as the implicit loop block of the iteration over all yielded values of the invocation, i. e., its contained statements are to be repeated. Subexpressions before the generator method invocation are evaluated once and stored in auxiliary local variables on the stack extension, and then the generator method is invoked:



The implementation of the `Routine` for the loop body contains the rest of the original expression tree. The already evaluated expressions are replaced by references to the auxiliary variables, the invocation of the generator method by a reference to the `value` parameter of the consumer method:



8.5 Bytecode Generation

The final task of the XL compiler is the translation of the expression tree representation to true class files for the Java virtual machine. We use the ASM library for this purpose [138]. This library facilitates the translation by providing a small but useful framework. It already includes the complete management of constant pools and structural information (e. g., class name, superclass name, signatures of declared fields and methods) of the class files to be generated, so it is easy to transfer the structural information from expression trees to class files. It remains to translate the method bodies from their intermediate representation as expression trees to bytecode instructions. Even here the ASM framework helps, since it automatically handles the computation of branch offsets for both forward and backward branches.

As has been stated above, the intermediate representation has been designed such that the translation to bytecode is mostly straightforward. The bytecode equivalent of the expression tree depicted in Fig. 8.3 on page 212 (see also its source code in 8.1 on page 209) is

```
bipush -42
invokestatic java/lang/Math.abs:(I)I;
invokestatic java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
astore_1
aload_0
aload_1
invokevirtual java/lang/Integer.intValue:()I
i2f
ldc 6.0f
fadd
f2d
putfield Test.y:D
```

The translation of this example and most other expressions of the XL programming language can be implemented by a post-order traversal which emits a suitable bytecode for each visited node. This procedure generates the postfix notation of the expression tree which has the right order for a stack machine like the JVM. However, control statements like loops require bytecode to be emitted before and after their children. Thus, a simple post-order traversal

is not sufficient for every node of the expression tree, the type of traversal depends on the specific class of a node. As a solution, the base class `de.grogra.xl.expr.Expression` declares the method

```
public void write(BytecodeWriter writer, boolean discard);
```

whose task is to write the bytecode equivalent of the expression node using the `writer`. By default, this method invokes itself on the children of the current node and then writes a class-specific bytecode operator (post-order traversal). This method can be overridden in subclasses in order to implement a different traversal.

8.5.1 Representation of Run-Time Models, Properties and Queries

In Sect. 6.5.1 on page 142 and Sect. 6.10.1 on page 174 run-time models for graphs and properties were presented which are identified by a name and which are obtained at run-time by the invocation of `modelForName` methods on corresponding factories. Conceptually, these models are core components of the run-time system of the XL programming language, but the Java virtual machine does not provide direct support for them (e. g., in the manner of the support for fields which are described in the constant pool of class files and referenced by bytecode instructions). Therefore, a compiler for the XL programming language has to generate suitable code to efficiently manage the models. It is clear that for efficiency run-time models should be obtained only once by invoking `modelForName` and then stored. Subsequent references to run-time models then only use the stored values. Basically, this is implemented by a static field `MODEL` with an initializer:

```
static final RuntimeModel MODEL
    = RuntimeModelFactory.getInstance().modelForName(name, loader);
```

Then `MODEL` can be used within code to reference the run-time model with name `name`.

However, if this code is part of the class in which the usage of the run-time model occurs, the run-time model is instantiated early, namely when the class is initialized. There are situations where it is required that the instantiation of the run-time model occurs lazily (as late as possible), i. e., when it is actually needed for the first time. For example, we might want to configure the factory in advance within the class which also contains the usage of the run-time model. Then that class is initialized before the configuration of the factory so that the latter occurs after instantiation of the run-time model and thus has no effect. As a solution to this problem, our compiler creates for each run-time model an auxiliary class which contains the above field declaration. Then the usage of the run-time model has to refer to the field `MODEL` in that auxiliary class. Now the lazy class initialization mechanism of the Java virtual machine guarantees the desired lazy instantiation.

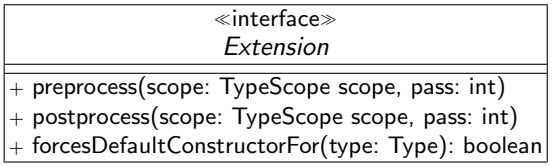
Within the auxiliary class of a run-time model for properties also all used properties of that run-time model are instantiated as part of the class initialization. I. e., for every used property, the auxiliary class contains a declaration

```
static final Property PROPERTYn = MODEL.propertyForName(name, loader);
```

Likewise, within the auxiliary class of a run-time model for graphs all queries belonging to it are instantiated and stored within static final fields. The representation of the query data within bytecode is by bytecode instructions, i. e., the bytecode contains (nested) constructor invocations for the query and the hierarchy of constituting patterns.

8.6 Compiler Extensions

The implemented XL compiler supports a mechanism for adding an extension which may modify the output of the compiler. Whenever a type declaration has an enclosing annotation `@de.grogra.xl.compiler.UseExtension`, the specified extension class is instantiated, and the compiler invokes the methods of the extension instance at certain points:



For each of the three passes of the semantic analysis (Sect. 8.3.1 on page 209) and each type for which the extension is active, `preprocess` is invoked before the compiler processes the type in that pass and `postprocess` is invoked afterwards. If `forcesDefaultConstructorFor` returns **true**, the compiler implicitly creates a default constructor without arguments.

8.7 Invocation of the Compiler

The XL compiler is included in the modelling platform GroIMP ([101], see also Appendix B). As it is common for integrated development environments, GroIMP implicitly invokes the compiler if a source file is saved. But the XL compiler can also be executed as a command line tool. For this purpose, the class `de.grogra.xl.compiler.Main` exists which can be invoked as a Java application. Its command line invocation is compatible with the `javac` compiler of Sun's Java Software Development Kit, i. e., it supports the standard command line options of the latter.

8.8 Current Limitations

As of GroIMP, version 0.9.8.1 [101], the XL compiler can not yet compile source code which makes use of generic types. As the mechanism for aggregate and filter methods as specified in Sect. 6.4.2 on page 138 and Sect. 6.3.5 on page 136 relies on generic types, the XL compiler currently implements a different mechanism, but this will be removed once a sufficient support for generic types is implemented.

8.9 Comparison with Java Compilers

Our implementation of an XL compiler is the only one which currently exists. Thus, in order to compare it with other compilers, we have to make use of the fact that the XL programming language is an extension of the Java programming language and restrict the comparison to the compilation of source code of the Java programming language. We compare our XL compiler (as contained in GroIMP, version 0.9.8.1; invoked as a command line tool) with three commonly used Java compilers: the `javac` compiler of Sun's Java Software Development Kit, version 1.6.0_03, the `jikes` compiler of IBM, version 1.22, and the compiler of the Eclipse Java Development Tools, version 3.2.1 (invoked as a command line tool and not as part of the Eclipse GUI). Possible criteria for a compiler comparison are the efficiency of the generated output, the required compilation time, the memory demand of the compiler, and the usefulness of error messages. Regarding the latter issue, we do not present an in-depth analysis, but simply mention that messages of the `jikes` compiler were used as a guideline which we found to be useful.

8.9.1 Efficiency of Output

Concerning the efficiency of the generated output, there is no major difference between the compilers. All four compilers do not optimize the bytecode at all. As an example, for the input

```
{
    int x = 1;
    x = 1;
    for (x = 1; x < 1; x++) {}
    int y = 0;
}
```

the XL compiler, IBM's `jikes` and the compiler of the Eclipse JDT produce exactly the same bytecode, namely

```
0:  iconst_1
1:  istore_0
2:  iconst_1
```

```
3:  istore_0
4:  iconst_1
5:  istore_0
6:  goto 12
9:  iinc 0, 1
12: iload_0
13: iconst_1
14: if_icmplt 9
17: iconst_0
18: istore_1
```

Sun's `javac` produces a slightly different bytecode for the loop:

```
0:  iconst_1
1:  istore_0
2:  iconst_1
3:  istore_0
4:  iconst_1
5:  istore_0
6:  iload_0
7:  iconst_1
8:  if_icmpge 17
11: iinc 0, 1
14: goto 6
17: iconst_0
18: istore_1
```

But as we can see, all tested compilers do not even apply simple optimizations: repeated assignments of the same value to a variable without usage of the value are not removed, loops which never get executed are not detected, local variable allocation does not reuse addresses whose previous content is no longer needed, local variables which are not used at all are not discarded. (There is one exception to the latter case: the Eclipse compiler can be configured such that unused local variables are discarded, this removes the last two instructions.) The similarity or even exact matching of bytecode among the four compilers holds not only for the presented case, but also in general. The reason for this is that there is a more or less straightforward translation from Java source code to bytecode, and that implementations of the Java virtual machine typically optimize at run-time so that optimizations of bytecode itself are not necessary.

Table 8.1 shows the number of class files and their total size for the four compilers if we use the source code of GroIMP, version 0.9.3.2 [101], as input, and if we choose compatibility with Java 1.4 for source code and bytecode. Note that we have to use this old version of GroIMP as the current one uses generic types, which the compiler can not yet handle. The source code consists of 2,586 files. The XL and Eclipse compilers produce the least number of class files. This is caused by their code for class literals (e. g., `String.class`), which never needs auxiliary class files. The other two compilers produce code which

needs an auxiliary class if class literals are used in the initializer of an interface. For the differences in the total size of all class files there are several reasons: the different treatment of class literals is one, another one is the option to perform inlining of instance initializers or finally-blocks.

Table 8.1. Size of compiled bytecode

Compiler	Number of Class Files	Total Size (Byte)
XL compiler	3,886	9,505,663
`javac`	3,905	9,532,771
Eclipse JDT	3,886	9,345,307
`jikes`	3,907	9,578,956

8.9.2 Efficiency of Compilation Process

To measure the efficiency of the compilation process, we used the same test case as before (source code of GroIMP, version 0.9.3.2) and measured the total compilation time and the peak demand on memory. We used a SUSE Linux 10.1 system equipped with a 2 GHz Intel Pentium 4M CPU and 1 GiB RAM. As Java runtime environment we chose Sun's JRE, version 1.6.0_03. Time measurements were done with the `time` command of Linux, where we used the sum of the user and system time as result. Memory measurements were done with the `top` command of Linux, where we used the peak physical memory as result (no swapping occurred during our tests). For the three compilers which are implemented in the Java programming language (XL, `javac`, Eclipse JDT), we set up the maximum heap size using the command line option `-Xmx` to a value such that the compilation time was not significantly increased compared to the setting with maximum possible heap size, but that further reduction of the heap size would significantly increase the compilation time (namely, due to an increased frequency of garbage collection). Thus, the demand on memory could be slightly reduced at the cost of compilation time. The results are shown in Table 8.2. As we can see, our compiler is least efficient with respect to both computation time and demand on memory. This is what we expected as efficiency issues played no major role in the development. For example, the high demand on memory results from the choice to have multiple passes which necessitates to keep the complete information of one pass in memory for the following pass. But the low efficiency is no significant problem as it is relatively unusual to compile 2,586 files simultaneously (e. g., GroIMP is split into projects which can be compiled separately if the compilation order respects project dependences). In particular, applications within the field of plant modelling are of much less size than GroIMP.

Table 8.2. Compilation time (average of 5 runs) and memory demand

Compiler	Compilation Time (s)	Required Memory (MiB)
XL compiler	85,5	511
`javac`	30,8	167
Eclipse JDT	22,4	85
`jikes`	10,9	105

The distribution of compilation time among the several phases and passes is as follows. The joint lexical and syntactical analysis are responsible for about 32% of the total time. The first pass of the semantic analysis consumes about 3%, the second pass about 8%. The final third pass, which constructs the expression tree and produces bytecode, requires about 57%.

Part III

Applications

In this part, we present a collection of rule-based applications of the XL programming language. At first, we describe an abstract base implementation of the graph and producer interfaces which will be used by all applications. Its graph rewriting mechanism conforms to relational growth grammars. It serves as a basis for an implementation with a minimalistic graph model, for an implementation which operates on XML DOM trees, for implementations for the scene graphs of the commercial 3D modellers 3ds Max and Maya, and for an implementation for the scene graph of the modelling platform GroIMP. The latter implementation is the most sophisticated one and is described in an own chapter together with applications from various fields.

9

Base Implementation and Its Applications

9.1 Base Implementation

The package `de.grogra.xl.impl.base` defines the abstract base implementation. It implements the XL interfaces for graphs and producers for a simple but efficient, still abstract graph model. The conformance with relational growth grammars, in particular the mechanism of parallel (two-level) derivations (Def. 5.16 on page 108), is achieved by the usage of modification queues which represent the current (still partial) parallel production.

9.1.1 Graph Model

The base implementation and all of its applications have a relatively simple abstract graph model. Nodes are objects in the sense of the Java programming language, where each concrete graph model specifies a base type for its nodes so that nodes may be instances of any subtype of this base type. Edges are directed, untyped, but attributed with a single **int**-value. Parallel edges are not allowed. The base implementation does not provide a means to restrict the allowed node types for an edge.

At first sight, this graph model differs from the one of RGG type graphs (Def. 5.3 on page 98): there, edges do not carry attributes, but they are typed; furthermore, the type graph may contain restrictions on the node types for a given edge type. Concerning the latter issue, it is left to concrete subimplementations of the base implementation to allow the specification of a sophisticated type graph and to perform corresponding integrity checks. Concerning the first issue, we may reinterpret the single **int**-valued attribute as a set of 32 independent bits which encodes the presence or absence of unattributed edges of 32 (concrete) edge types. As bitwise operations are in natural correspondence to set-theoretic operations, this gives rise to an efficient representation of edges and abstract and concrete edge types. For example, an abstract edge type of an RGG type graph (see Def. 5.3 on page 98) is simply represented by

an **int**-value having more than one bit set (one bit for each concrete subtype). Or, an RGG edge of a given edge type T exists from a node to another if there is an edge with bits b such that the bitwise 'and'-operation $b \,\&\, T$ yields a non-zero value.

For practical purposes, the edge encoding is a bit more complex: the higher 24 bits are exactly interpreted as described above, i. e., they independently switch 24 normal edge types on or off, but the lower 8 bits are interpreted as a single number that, if non-zero, represents one of 255 special edge types which, thus, are mutually exclusive.

Having only 32 bits to encode edge information between an ordered pair of nodes is of course a restriction, but for all applications which have been considered so far, this did not pose any problem. The main advantage that outweighs this restriction is the efficient representation: as it has been mentioned in Sect. 5.2.2 on page 97, most edges in typical applications have types drawn from a small set; for these edges the bit-encoding is very well suited and memory-efficient. In case it does not suffice, one can make use of auxiliary nodes which play the role of edges. This is also the standard solution if attributed edges are required, see the discussion in Sect. 5.2.2.

The graph model is reflected in the implementation of the classes and methods. Figure 9.1 shows a class diagram for the implementation of the graph-related XL interfaces, a more detailed description can be found in the API documentation [101].

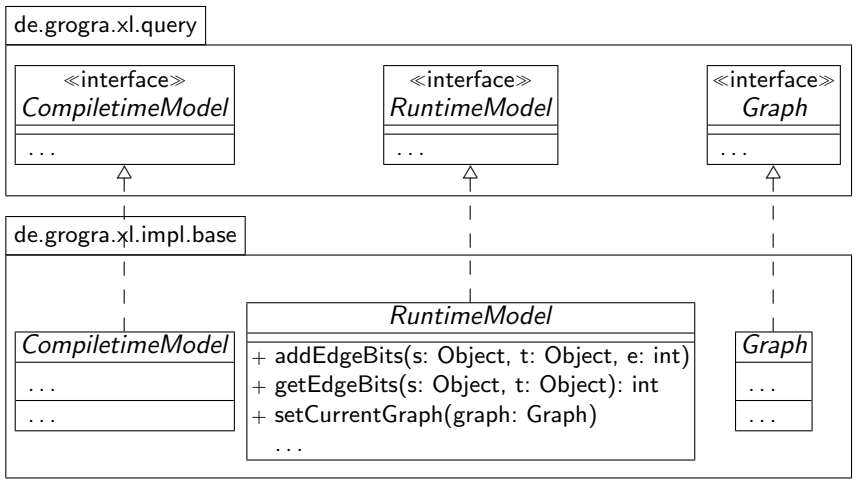


Figure 9.1. Class diagram for base implementation

The class `RuntimeModel` also declares a collection of **int**-valued constants which represent predefined edge types. The constants `SUCCESSOR_EDGE` and `BRANCH_EDGE` encode the relations known from L-systems and axial trees (see

Sect. 5.2.1 on page 95). The edge type `REFINEMENT_EDGE` is associated with edge symbols like `/>`, `</` following a notation for multiscaled structures in [67]. The constants `CONTAINMENT_EDGE`, `CONTAINMENT_END_EDGE` and `MARK_EDGE` are used for the representation of interpretive structures within the graph (see Sect. 9.1.6 on page 248). Finally, the constant `MIN_USER_EDGE` is the minimal edge type which has no predefined meaning and can be used freely by the user. Here, 'minimal' means minimal as a bit mask, i. e., all other unused edge types can be obtained by shifting the bits of `MIN_USER_EDGE` to the left.

9.1.2 Modification Queues

The XL programming language in itself does not define what exactly happens when a match for the left-hand side of a rule has been found and the right-hand side is executed. This is completely the responsibility of the implementation of the producer which is used for the execution of the right-hand side, see Sect. 6.8.2 on page 173. On the other hand, for relational growth grammars we explicitly demand a parallel mode of rule application based on the single-pushout approach, see Sect. 5.5 on page 106. Thus, for relational growth grammars the producer has to implement such a parallel mode.

Given that the XL programming language has a conventional sequential control flow, we have to simulate the parallel mode. The discussion about RGG derivations in Sect. 5.5 already gives a hint how to do this: a parallel RGG derivation can be seen as a two-level derivation where the first level combines elementary SPO productions and connection edges into a single parallel production and the second level performs a direct sequential derivation using the final parallel production. So if the execution of a right-hand side does not modify the graph directly, but only adds some entries to a representation of the parallel production to be constructed, we have an implementation of the first level. The second level then consumes all collected entries, thus applying a derivation using the parallel production. Note that this application is not invoked by the rules themselves, whose sole effect is the addition of entries, but by some external mechanism (e. g., an invocation of a dedicated method).

As a representation of the parallel production, it turns out that a collection of *modification queues* is suitable. In order to be able to add an elementary SPO production $L \hookrightarrow R$, we need four types of queue entries:

1. *Addition of nodes.* Each node of R which is not contained in L, i. e., each new node, leads to such an entry.
2. *Addition of edges.* Likewise, each new edge of R has to be represented as a corresponding entry.
3. *Deletion of nodes.* Nodes of (the match of) L which are not part of R, i. e., which shall be deleted, are recorded in a deletion entry.
4. *Deletion of edges.* Likewise, edges which shall be deleted are recorded.

Then, to perform the derivation using the combined parallel production we have to execute the actions associated with the entries, thereby modifying the

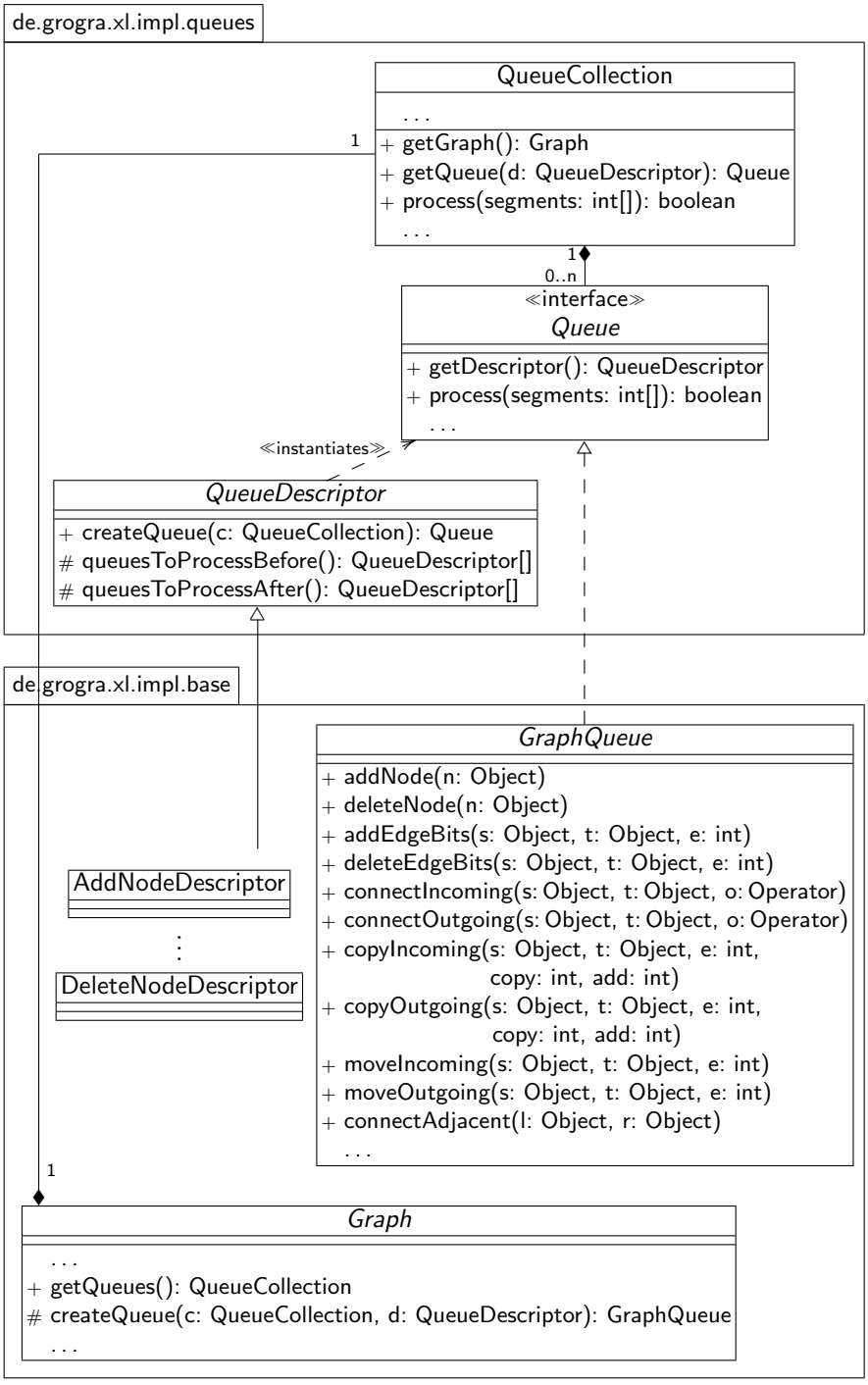


Figure 9.2. Class diagram for modification queues

current graph from the host graph to the derived graph. At first, all entries which add nodes have to be processed, then all entries which add edges, then all edge-deleting entries and finally all node-deleting entries, where the deletion of a node also removes all dangling edges. This order ensures the correct SPO behaviour: as deletions are executed at last, they may undo previous additions of edges and are thus favoured. An efficient way to ensure this order of processing of entries is to use a collection of queues which contains an own queue for each type of queue entries. Then the queues have to be processed in the correct order, while the order within a single queue is irrelevant (we use first-in-first-out queues). To include the mechanism of connection transformations, we insert a further *connection queue* between the node- and edge-adding queues. Its entries specify individual connection transformations which potentially lead to further additions of edges.

The implementation of such a collection of queues is split into two parts, see the class diagram in Fig. 9.2 on the preceding page. The first part is contained in the package `de.grogra.xl.impl.queues` and is independent of the the rest of the base implementation, i. e., it can also be used for other graph models or the modification of properties in parallel, see Sect. 9.1.8 on page 251. This part contains a `QueueCollection` and an interface `Queue` together with an abstract class `QueueDescriptor` which serves as a factory for queues and also specifies the order of processing of queues. The invocation of the method `process` on a queue collection invokes `process` on all of its queues, where the order of invocation conforms to the specifications made by the methods `queuesToProcessBefore` and `queuesToProcessAfter` of the descriptors.

The package `de.grogra.xl.impl.base` contains an abstract queue implementation `GraphQueue`. Its methods like `addNode` and `addEdgeBits` append entries to the queue, where the concrete data (e. g., **int** values for edges) is specific to the graph model of the base implementation. For each required queue, there is a dedicated descriptor class (e. g., `AddNodeDescriptor`). Such a class delegates the instantiation of its queue to the method `createQueue` of the associated graph; this abstract method then has to return a suitable concrete queue. The graph of the base implementation also maintains a queue collection and provides the method `derive` to trigger the processing of all queues. I. e., this method marks the end of the current parallel two-level derivation and performs a derivation using the constructed parallel production.

The methods `connectIncoming` and `connectOutgoing` of `GraphQueue` add connection transformations entries to the queue. The methods `copyIncoming`, `copyOutgoing`, `copyIncoming`, `copyOutgoing`, and `connectAdjacent` serve a similar purpose. This is explained in the next section.

9.1.3 Implementation of Connection Mechanism

A connection transformation of relational growth grammars (Sect. 5.5 on page 106) of the form $(s, (A, d, \gamma, h), t)$ with a node s from the left-hand side

of a rule, a node t from the right-hand side, an operator A, a direction d, a concrete edge type γ and a flag h is added to the current parallel production by invocation of `connectIncoming` or `connectOutgoing` (depending on d) on the connection queue. These methods have three parameters, namely for s, t and an operator A which indirectly specifies edge type γ and flag h by the methods shown in Fig. 9.3.

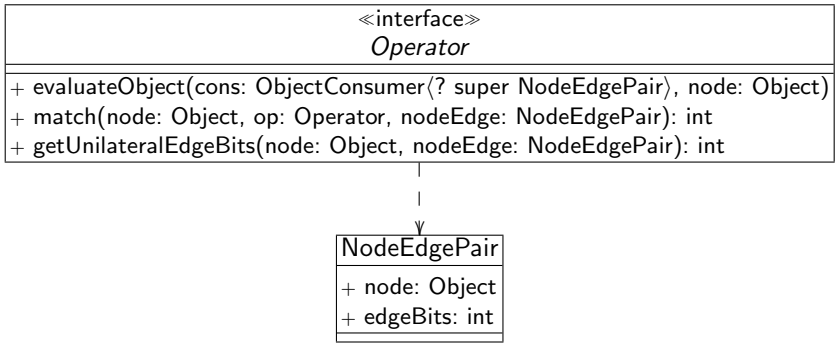


Figure 9.3. Class diagram for operators

When the connection queue is processed, for each entry (s, t, A) the method `evaluateObject` is invoked on the operator A with s as `node`-parameter. This yields all nodes $s' \in A_G(s)$ together with a concrete edge type γ to the specified consumer `cons`, where (s', γ) are stored in a `NodeEdgePair`. In fact, γ may have more than one bit set, which then stands for a whole set of concrete edge types for potential connection edges. Now given s', for any potentially matching entry (s', t', A') with the opposite direction the method `match` is invoked on A' with the arguments $(s, A, (s', \gamma))$. If the type (or set of types) γ' for connection edges potentially created by A' does not overlap with γ, the connection transformation has no matching edge type, and the method `match` returns 0. Otherwise, at least edge type and direction of both connection transformations match. If furthermore $s \in A'_G(s')$, both connection transformations match completely, and a connection edge between t and t' is created whose type (or set of types) is determined by the returned value of `match`. Otherwise, only edge type and direction match, then the special value `ONLY_EDGE_TYPE_MATCHES` is returned.

Now if for an entry (s, t, A) and a node $s' \in A_G(s)$ all invocations of `match` for entries (s', t', A') with the opposite direction return 0, there is no connection transformation for s' and a matching edge type. Then the method `getUnilateralEdgeBits` is invoked on A with the arguments $(s, (s', \gamma))$ to determine the type (or set of types) of the connection edge to be created between s' and t. Thus, the method `getUnilateralEdgeBits` is responsible

for the implementation of the h-flag of an RGG rule: if this is 0, the method has to return 0 to prevent the creation of connection edges with only one involved connection transformation.

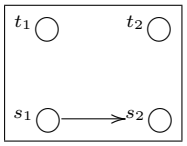
Given A, γ, h of a connection transformation of an RGG rule, the interface `Operator` would be implemented as follows:

```
class Op implements Operator {
    public NodeEdgePair* evaluateObject(Object x) {
        yield (A_G(x), γ);
    }

    public int match(Object node, Operator op, NodeEdgePair nodeEdge) {
        if ((γ & nodeEdge.edgeBits) != 0) {
            // non-empty intersection of sets of edge types
            if (node ∈ A_G(nodeEdge.node)) {
                return γ & nodeEdge.edgeBits;
            } else {
                return ONLY_EDGE_TYPE_MATCHES;
            }
        } else {
            return 0;
        }
    }

    public int getUnilateralEdgeBits(Object node, NodeEdgePair nodeEdge) {
        return (h == 1) ? γ : 0;
    }
}
```

The class `de.grogra.xl.impl.base.Neighbor` implements `Operator` for the operator N^d_γ. It makes use of the possibilities of the presented mechanism to represent all connection transformations required for a single node of a right-hand side of a translated L-system production (Def. 5.6 on page 100) by a single entry in the connection queue.

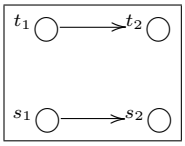
The determination if a connection edge shall be created involves simultaneous knowledge of the connection transformations of both s and $s' \in A_G(s)$. Regarding an efficient processing, it would be preferable to process each connection transformation individually. This way, we cannot obtain an implementation of connection transformations which fully conforms to theory, but for the operators N^d_μ there is a solution which "sufficiently" conforms to theory for translated L-system productions. Namely, a connection transformation $(s, (N^d_\mu, d, \gamma, h), t)$ creates, for every μ-typed edge of s with direction d, a γ-typed edge between the adjacent node s' and t with the same direction with respect to s'; all connection transformations are processed sequentially as part of the connection queue. As an example, consider the following setting, where $*$ denotes the single edge type:

$$G = \quad , \qquad Z = \left\{ \begin{array}{l} C_1 = (s_1, (N_*^{\text{out}}, \text{out}, *, 1), t_1), \\ C_2 = (s_2, (N_*^{\text{in}}, \text{in}, *, 1), t_2) \end{array} \right\}$$

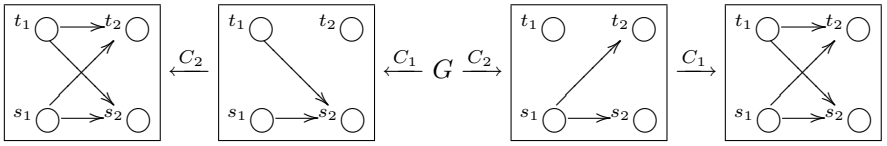


According to theory, the two connection transformations of Z would establish a connection edge from t_1 to t_2:

$$G' = \quad .$$



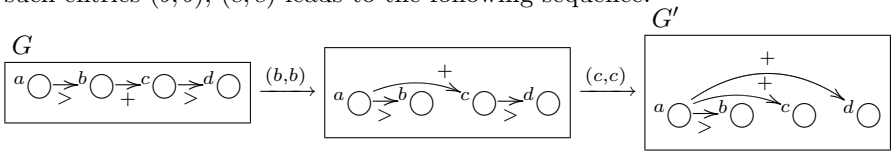
The sequential application could of course depend on the order of processing of connection transformations. In this case, it does not:



As we can see, the result of the sequential application contains the desired connection edge, but also two additional edges. However, if the predecessor nodes s_1, s_2 are deleted as part of the rule, these additional edges are short-lived, and the final result coincides with the theory.

This variant of the connection mechanism is implemented by the methods `copyIncoming` and `copyOutgoing` of `GraphQueue`, see Fig. 9.2 on page 238. The methods `moveIncoming` and `moveOutgoing` are similar, but delete the original edge.

The important method `connectAdjacent` adds an entry to the connection queue which can be used to handle non-axis-propagating L-system productions. When such an entry with the parameters (l, r) is processed, for any incoming successor or branch edge (s, γ, l) of l and any outgoing successor or branch edge (r, μ, t) of r an edge from s to t is created. It is a branch edge if at least one of γ, μ is a branch, otherwise it is a successor edge. Furthermore, all outgoing edges (r, μ, t) are deleted. As an example, the application of two such entries (b, b), (c, c) leads to the following sequence:



If we reversed the order, the final result would be the same. The effect is that the adjacent nodes of (l, r) become neighbours as if we had removed the symbols B, C from the word AB[CD], resulting in A[D], and considered the corresponding graph. But the nodes b, c are still in the graph so that connection

transformations, which take effect after entries created by `connectAdjacent`, can use their incoming edges. For example, for the non-axis-propagating L-system productions $\mathsf{B} \to [\mathsf{E}]$, $\mathsf{C} \to \varepsilon$ and the word $\mathsf{AB}[\mathsf{CD}]$, the result is $\mathsf{A}[\mathsf{E}][\mathsf{D}]$. The corresponding derivation of the graph G at first moves edges as above, resulting in G', then applies the connection transformations of $\mathsf{B} \to [\mathsf{E}]$, which create a branch edge from a to the new E-node due to the successor edge from a to b, and finally removes b, c. The result is the correct graph representation of $\mathsf{A}[\mathsf{E}][\mathsf{D}]$.

9.1.4 Producer Implementation

The class `de.grogra.xl.impl.base.Producer` is an abstract implementation of the producer interface of XL (Sect. 6.8.2 on page 173). All methods specified by this interface are implemented, but methods for prefix operators of production statements (Sect. 6.7.3 on page 166) are not provided. Instead of this, there are several useful methods to which such operator methods can delegate:

Producer
pushImpl() # popImpl() # separateImpl() # addNodeImpl(node: Object, addEdge: boolean) # addEdgeImpl(first: Object, second: Object, bits: int, dir: EdgeDirection) # nodeUsed(node: Object)

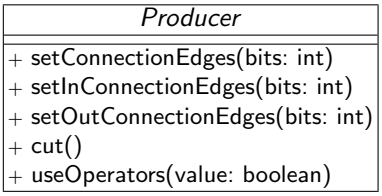
The first three methods provide the basic behaviour needed for the implementation of the producer methods `producer$push`, `producer$pop` and `producer$separate` which were discussed in Sect. 6.7.4 on page 168. They modify the internal state of the producer, which consists of the previous node and the default edge type for the next edge. For example, `pushImpl` pushes this state onto a stack and sets the default edge type to `BRANCH_EDGE`. This ensures that a branch edge is used by default after an opening bracket.

The method `addNodeImpl` uses this internal state: if the flag `addEdge` is **true** and there is a previous node, then an edge with default type from the previous node to the new `node` is created, the previous node is updated to be `node`, and the default edge type is set to `SUCCESSOR_EDGE`. This already ensures that for production statements `a [b c] d` a branch edge from `a` to `b` and a successor edge from `b` to `c` and from `a` to `d` is created. To be more precise, an edge is not immediately created, the producer just registers that an edge shall be created afterwards, i. e., when the derivation is performed. Furthermore, `addNodeImpl` invokes `nodeUsed` which informs the producer that a node appeared on the right-hand side. This is important for nodes which stem from the left-hand side as it tells the producer that these are reused on the right-hand side and, thus, must not be deleted within the SPO derivation. In addition, `nodeUsed` appends an entry to the node-adding queue. On invocation

of the method `addEdgeImpl`, the producer registers that an edge of specific type and direction between the given nodes shall be created afterwards.

The implementation of the method `producer$beginExecution` specified by the common producer interface mainly serves to control the derivation mode by its **boolean** return value, see Sect. 9.1.5 on the facing page. The method `producer$endExecution`, which is invoked at the end of the execution of a right-hand side, has a more complex task. At first, it visits the current match in order to receive information about matched non-context edges, see Sect. 7.6 on page 201. Then, in order to ensure the correct SPO behaviour, it adds for each edge of the right-hand side which does not exist as a matched non-context edge an entry to the edge-adding queue, and for each matched non-context edge without counterpart on the right-hand side an entry to the edge-deleting queue.

The next actions of `producer$endProduction` depend on the derivation mode. The normal behaviour is to append an entry to the node-deleting queue for each matched non-context node which is not reused on the right-hand side, and, if the used rule arrow is `==>`, to append entries to the connection queue. The latter entries are controlled by the following methods:

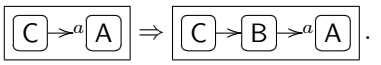


By default, a connection transformation entry (s, t, A) is added for the treatment of the L-node in Def. 5.6 on page 100 with s being the match of the textually leftmost node pattern (the node returned by the invocation of `getInValue` on the query state, see Sect. 7.6 on page 201) and t being the first node of the right-hand side which is not in brackets and not behind a separating comma. A similar entry is responsible for the rightmost node pattern and the last node of the right-hand side (or the first part thereof if a comma is used as separator), this implements the treatment of the R-node in Def. 5.6. Furthermore, for each node of (the first part of) the right-hand side which is in brackets, but has no previous node, we append an entry corresponding to the handling of B-nodes in Def. 5.6. Normally, these entries copy all edges from old to new nodes, but the types of copied edges can be restricted by the above listed methods in the way indicated by their name. The invocation of the method `cut` is equivalent to `setOutConnectionEdges(0);`, i. e., no outgoing edges are copied. This can be used to implement the cut-operator % of L-systems. As an example, the rule `A B ==> [[C] D] E, F;` has connection transformations for all incoming edges from the `A`-node to the `E`-node, for all outgoing edges from the `B`-node to the `E`-node, and for all incoming edges from the `A`-node to the `C`- and `D`-nodes, but changing the type to branch edges. The rule `A ==> B cut;` only has connection transformations for incoming edges from

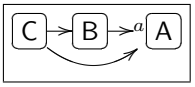
the `A`-node to the `B`-node (recall that methods of the producer can be invoked by their simple name, see Sect. 6.7.1 on page 164).

If the right-hand side of a `==>`-rule has no leftmost node (e. g., in case of a non-axis-propagating L-system production), an additional entry is added to the connection queue by the method `connectAdjacent` with the matches of the textually leftmost and rightmost node patterns as arguments. As we have desribed in the previous section, this ensures a suitable behaviour for such cases.

Normally, `==>`-rules delete their predecessor. But this is not mandatory and we may write a rule like `a:A ==> B a`. The intended meaning of such a rule is that an `A`-node creates a `B`-node and inserts this node below itself as in the sequence



But with the mechanism described so far, we would get a different final graph:



Therefore, for any node which is not deleted and for which a connection transformation is added, all existing edges which match the connection transformation are deleted as part of the edge-deletion queue. This yields the desired derivation.

By invoking the method `useOperators` with a **false** argument, the mechanism based on the methods `copyIncoming` and `copyOutgoing` is used instead of the default operator-based mechanism using `connectIncoming` and `connectOutgoing`, see the previous section.

If the flag `INTERPRETIVE_FLAG` is set for the derivation mode, nothing is added to the node-deleting queue, and entries to embed the right-hand side as interpretive structure are appended to the connection queue. This is discussed in more detail in Sect. 9.1.6 on page 248.

9.1.5 Derivation Modes

Rules of the XL programming language are executed when the control flow reaches them. The execution of a rule finds all matches of the left-hand side and, for each match, performs the statements of the right-hand side. Thus, for a producer implementation which constructs a parallel production, the control flow in the sense of relational growth grammars (Def. 5.20 on page 112) uses the family of all executed rules and applies each rule via every possible match in the current graph. This is a true parallel mode of derivation.

If one wants to restrict the actually used rules or their matches, there are two possibilities: either one has to add corresponding logic for the control of rule application to the program itself, or the implementation of the XL interfaces provides some common mechanism. The first possibility is of course

the most general one, but there are some usual restrictions of the used matches which can be controlled by the implementation of the XL interfaces:

- A *sequential mode* of derivation selects a single rule with a single match per step. The selection could be *deterministic* (e. g., the first found match) or *nondeterministic* (pseudorandom choice among all matches of all rules).
- A *nondeterministic parallel mode* extends the behaviour of nondeterministic L-systems to parallel graph grammars: if several matches would delete the same node, all but one are discarded on the basis of a random choice.

A problem in choosing rules and matches is that at no point of execution the set of available rules is known. A rule simply is available if it is reached by the control flow of the Java virtual machine, and when the control flow leaves it, the rule is forgotten. The only knowledge about the rule consists in side-effects of its execution, namely the addition of queue entries in case of the base implementation.

Fortunately, this knowledge suffices to implement the above listed modes of derivation. Namely, for the deterministic sequential mode which uses the first found match we have to remember whether we have already found a match. If so, further matches have to be ignored, which can be effected by the producer if it returns **false** on invocation of its method `producer$beginExecution`, see Sect. 6.8.2 on page 173. Of course, this is a very inefficient implementation as all remaining rules are still executed, i. e., it is still looked for all matches of their left-hand sides, although we know that we do not need these matches.

For the nondeterministic sequential mode, each found match is a potential candidate for the actually used match. A reasonable nondeterminism is to randomly choose the actual match among all matches with equal probability. Thus, if there are n matches in total, the probability of a single match is $\frac{1}{n}$. But we do not know n in advance. Fortunately, there is a simple solution which does not require the knowledge of n. Namely, the first found match is preliminarily taken as actual match, i. e., the corresponding right-hand side is executed and leads to entries in the modification queues. Then, if there is a second match, with a probability of $\frac{1}{2}$ this match is taken as new preliminary match by removing the previous entries from the queues and executing the right-hand side for the second match, leading to new entries. Otherwise, the second match is ignored. This procedure continues until all matches have been processed, using a probability of $\frac{1}{i}$ for match i to replace the previously used match. The probability of match i to be the final actual match is as desired:

$$\frac{1}{i} \prod_{j=i+1}^{n} \frac{j-1}{j} = \frac{1}{n}.$$

For the nondeterministic parallel mode, right-hand sides are executed for each match. In addition to the default deterministic mode, the queues are partitioned into segments, one segment for each match. When the queues are processed, a random order for the segments is chosen, and the segments are

processed in this order. But a segment is excluded if it would delete a node which has already been deleted by a previous segment. If the application of each individual match deletes at most a single node, this corresponds to an equal probability for each match within a group of matches which delete the same node. However, as left-hand sides may contain more than one node, the application of a match may delete more than one node and the sets of deleted nodes of matches may overlap in an arbitrary way. For example, consider the case that the application of a match m_1 deletes two nodes a, b, the application of a match m_2 deletes a and the application of a match m_3 deletes b. Then, if m_1 is applied as first, the other ones are excluded. If m_2 or m_3 is applied as first, m_1 is excluded.

The implementation of both nondeterministic modes is supported by the possibility to partition the queues into segments which are addressed by values of type **int**. The related methods are shown in Fig. 9.4. At first, the start of a new segment has to be externally triggered by invocation of `startNewSegment` on the queue collection. For the sequential mode, this happens for the first match, while for the parallel mode, this happens for each match. Then, for the sequential mode `resetToSegment` is invoked for each new preliminary match in order to clear any previous data (resulting from the old preliminary match) in the segment. The parallel mode is more complicated. Firstly, the `process`-method of the queue collection is invoked with a random order of the segments (specified as argument to the parameter `segments`). This is the order in which the segments shall be processed, but before doing so, the queue collection invokes `clearSegmentsToExclude` on each of its queues to give them an opportunity to remove segments from the order. This is actually used by the node-deleting queue, it removes all segments which delete nodes that are also deleted by previous segments. Finally, `process` is invoked on each queue with the modified order.

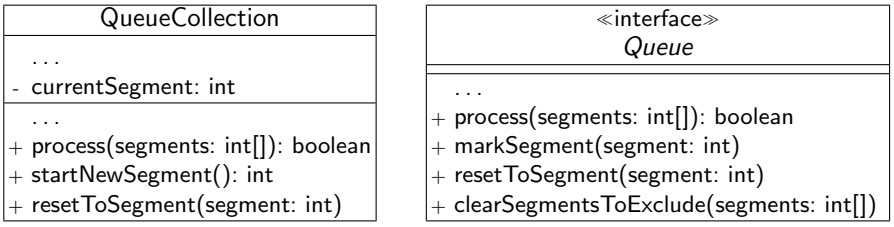


Figure 9.4. Class diagram for segmentization support of queues

To choose the derivation mode, the class `Graph` of the base implementation provides the method `setDerivationMode` and a set of constants to use as argument, namely `PARALLEL_MODE`, `PARALLEL_NON_DETERMINISTIC_MODE`, `SEQUENTIAL_MODE` and `SEQUENTIAL_NON_DETERMINISTIC_MODE`. These constants

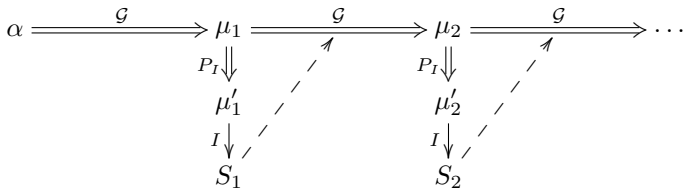
can be combined with two flags as modifiers, `EXCLUDE_DELETED_FLAG` and `INTERPRETIVE_FLAG`, as in

```
graph.setDerivationMode(PARALLEL_MODE | EXCLUDE_DELETED_FLAG);
```

This is also the default derivation mode. The flag `EXCLUDE_DELETED_FLAG` is useful in combination with `PARALLEL_MODE` to obtain a variant of the deterministic parallel mode where a node is deleted at most once. Namely, when the execution of the right-hand side for a match leads to the deletion of a node, this is remembered, and this node is excluded from the set of candidates for further matches by the mechanism described in Sect. 7.3.4 on page 199 (i. e., the method `excludeFromMatch` of the query state returns **`true`** for such nodes). In effect, this means that the first match which deletes a node is used and all further possible matches are disabled. Concrete examples for all described derivation modes can be found in Sect. 10.2.1 on page 278.

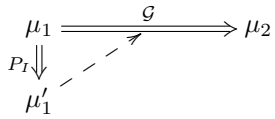
9.1.6 Interpretive Structures

The modifier `INTERPRETIVE_FLAG` for the derivation mode activates a special embedding mechanism which emulates interpretive productions as known from L-systems (Sect. 3.9 on page 28). Figure 3.6 on page 31 shows the derivation mechanism of L-systems if sensitivity is included, we repeat it here:



The set P_I of interpretive productions derives the original word μ to another word μ' which is then interpreted to a geometric structure S by the turtle I. The generative productions in $\mathcal{G}$ can make use of this structure when deriving a new generation.

Unfortunately, this mechanism does not fit well to our graph-based setting. Firstly, we have true scene graphs in mind. They represent geometric structures by themselves so that there is no turtle interpretation and no additional structure S. This would lead to a mechanism



.

But this requires two graphs μ, μ' at the same time, which wastes time and space for large graphs, or some representation of the changes made by P_I within μ. Furthermore, in this setting sensitive rules of $\mathcal{G}$ can not always be stated in a natural way. For example, think of a rule which needs access to the next node of type `F` on the path to the root:

```
Bud (* (<--)+ : (f:F) *), (!isShadowed(f)) ==> Internode Bud;
```

Now if this F-node is the result of an interpretive rule

```
Internode ==> F(...);
```

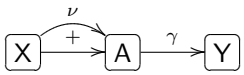
the matching of the pattern `Bud (* (<--)+ : (f:F) *)` would have to allow only nodes from μ for the pattern `Bud`, but has to include also nodes from μ' for the context pattern.

Therefore, we change the mechanism to an alternating application of the rules of $\mathcal{G}$ and P_I:

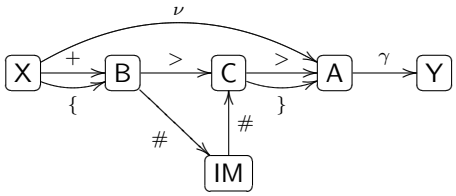
$$\mu_1 \overset{P_I'}{\Longrightarrow} \mu_1' \overset{\mathcal{G}'}{\Longrightarrow} \mu_2 \ .$$

Actually, we use modified versions $\mathcal{G}'$ and P_I'. $\mathcal{G}'$ contains the rules of $\mathcal{G}$, but also a mechanism to remove all structures created by interpretive rules. In order for this to be possible, the latter must not delete anything, and they must provide information in the graph where and how new objects were added. This information is set up by the modified version P_I' of the interpretive rules. The precise procedure is as follows. Firstly, only rules of L-system type are allowed as interpretive rules, i. e., the (non-context part of) the left-hand side must consist of a single node which is replaced by the right-hand side which consists of new nodes only. For example, consider the interpretive rule

Now if there is an A-typed node in the graph as in



the application of the interpretive rule does not really replace this node, but prepends its right-hand side in front:



Incoming edges of type branch or successor are moved from the A-node to the new B-node, all other edges are left unchanged (the ν, γ-typed edges in the example). Furthermore, a successor edge from the C-node to the A-node is created. Finally, the new interpretive structure is marked as such in two ways:

1. A node of a special type IM (which stands for 'interpretive mark') is created. Its two incident edges of type 'mark' (symbolized by #, internally

represented by the constant `MARK_EDGE`, see Sect. 9.1.1 on page 235) point to those nodes of the interpretive structure which are the connectors to the outer graph. This helps to identify the locations of interpretive structures when the latter shall be removed.

2. The two edges which establish the links between connectors and the outer graph are supplemented with edges of type 'containment' (symbolized by {, internally represented by `CONTAINMENT_EDGE`) and 'end of containment' (symbolized by } and internally represented by `CONTAINMENT_END_EDGE`). These edges help to get from connectors to the outer graph. The usage of the term 'containment' is motivated by the typical usage of interpretive rules to specify which detailed geometric components an entity of the model contains.

This is not equivalent to interpretive productions in L-systems: the A-node is still there, while it is not contained in the interpreted L-system word, and the local context of the A-node, which could be used in a generative rule, is changed. On the other hand, the solution has several advantages.

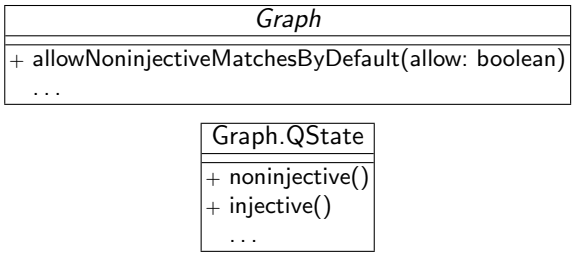
- It is simple to implement and efficient with respect to the memory consumption as there is no need to create a copy of the whole graph.
- Interpretive structures are visible within generative rules. This is important for global sensitivity if, for example, this tests for some geometric conditions, but the geometry is only defined by interpretive rules.
- Interpretive structures are immediately in front of their inducing node. This helps to implement the sensitive functions of the GROGRA software (Sect. 3.15.1 on page 35) which are usually defined for the "current unit" which is the last created shoot, including shoots created by an interpretive production for the current symbol. For example, if the current word is Fa, the F symbol is the current unit when a sensitive function within a production for a is invoked. However, if there is an interpretive production like $a \rightarrow F$, the latter F is taken as the current unit. In our setting, both cases can be handled by traversing the graph downwards to the root until the first shoot node is found.
- Interpretive structures may have an influence on the subsequent structure. For example, the L-system word aF with the interpretive production $a \rightarrow RU(45)$ is drawn as a cylinder with a prepended rotation of 45 degrees. The same effect is achieved by our setting if we interpret the graph as a 3D scene graph.

If the current derivation mode is interpretive, the special embedding of right-hand sides is performed as part of the connection queue. To add corresponding entries, the method `embedInterpretive` is declared in the class `GraphQueue`.

9.1.7 Injectivity of Matches

By default, matches for nodes of a query have to be injective, see Sect. 7.3.4 on page 199. This is controlled by the method `allowsNoninjectiveMatches`

of the `QueryState`. The base implementation overrides this method in the class `Graph.QState` order to be able to turn off the enforcement of injectivity. Three methods control the behaviour:



The first one sets the behaviour for all queries subsequently executed within the graph. The two methods of `QState` control the behaviour of the current query and can be invoked at the beginning of a query as in the following code (see also Sect. 6.5.10 on page 155):

```
(* (noninjective()) a:A, b:A *) // also a == b is a match
```

9.1.8 Implementation of Properties

The package `de.grogra.xl.impl.property` contains abstract implementations of the compile-time and run-time models for properties (Sect. 6.10.1 on page 174). These implement only very basic methods of the model interfaces, most interesting is a collection of subinterfaces of `RuntimeModel.Property` as shown in Fig. 9.5 on the following page. For each primitive type and `Object`, there is a specialized subinterface which declares methods for deferred assignments (see Sect. 6.10.3 on page 177) for those operators which have a natural meaning for the type. For example, for numeric types there are methods for the operators `:=`, `:+=`, `:-=`, `:*=`, `:/=`.

The base implementation does not provide an implementation of these property interfaces. However, the provided general mechanism of modification queues is useful for the implementation of deferred assignments. I. e., in concrete implementations, the operator methods for deferred assignments could add modification entries to a corresponding queue. If this queue is processed together with the other queues for structural changes, we achieve a parallel derivation which includes both structural and internal changes. Entries of this queue which were created by compound deferred assignments like `:+=` can be seen as representations of incremental modification edges which were introduced in Sect. 5.8 on page 115, while processing these entries corresponds to the sequential cumulation of these edges into the property value.

9.2 Simple Implementation

The package `de.grogra.xl.impl.simple` contains a simple, mostly concrete implementation on top of the base implementation. Its central class and at

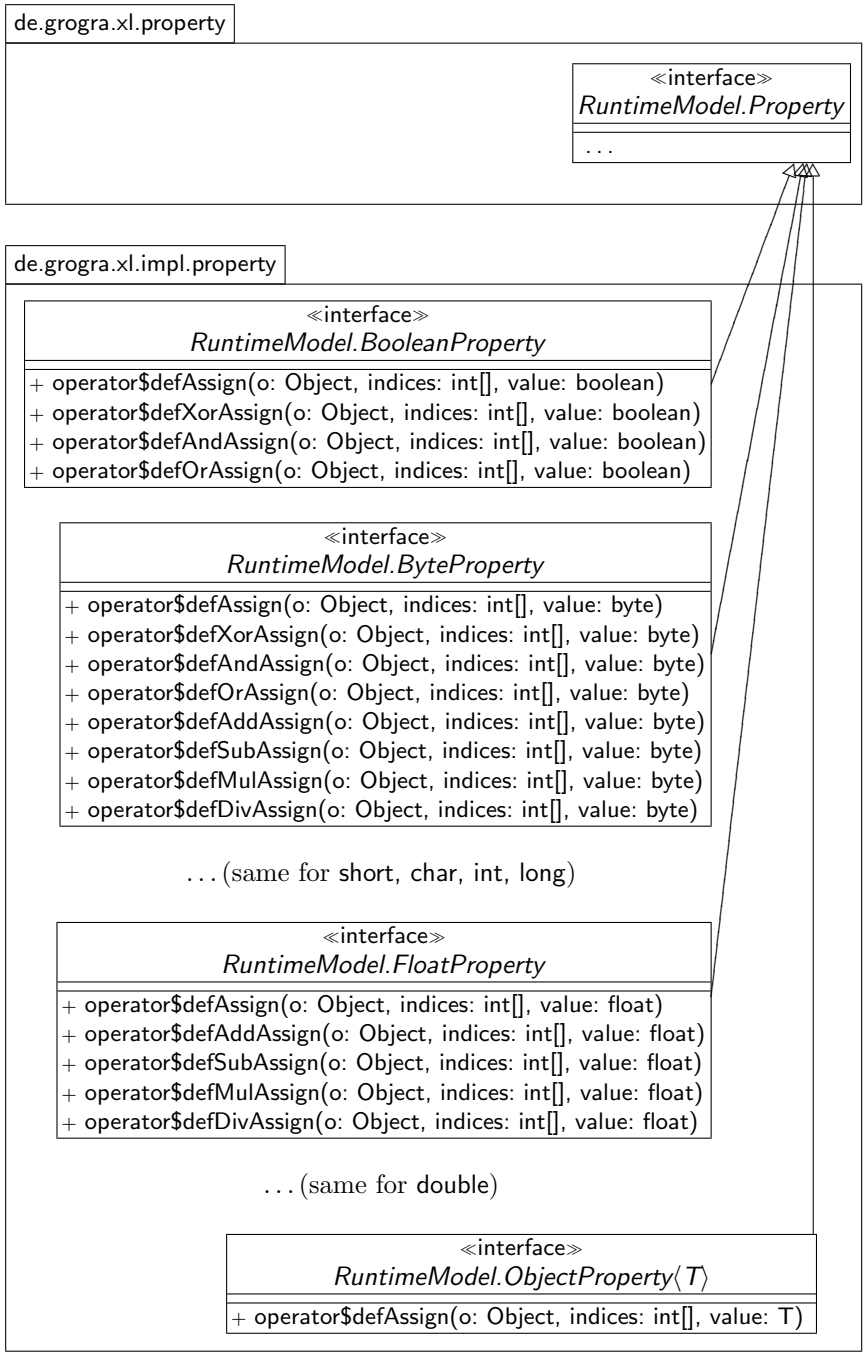
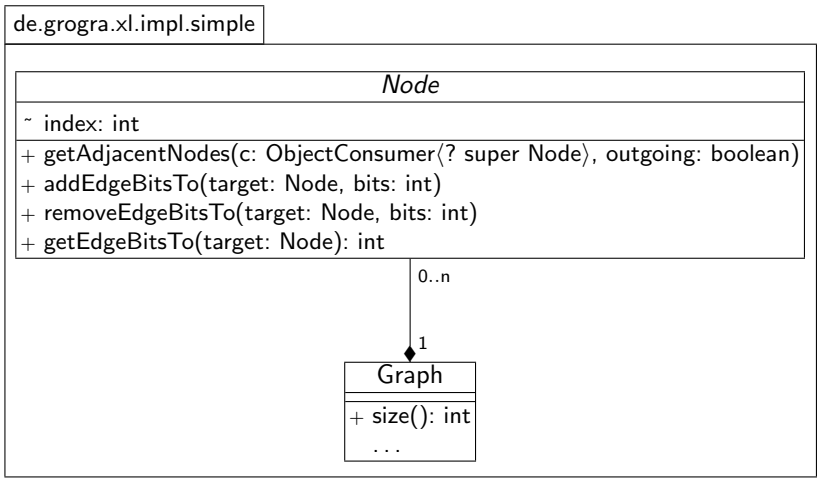


Figure 9.5. Class diagram for property subinterfaces

the same time the only abstract class is the class `Node`, which represents nodes of the graph and declares four abstract methods:



These methods serve to inspect and modify adjacent nodes of the current node, and the `Graph` class uses them to implement all the abstract methods specified by its supertypes. I. e., how edges are represented is completely left to the implementation of the methods of `Node` in subclasses. The advantage of this is the possibility of very lightweight representations of graphs.

9.2.1 Sierpinski Triangles

Such a lightweight representation is especially important if large graphs of millions of nodes have to be represented and modified. The Sierpinski case study of the AGTIVE '07 conference is a typical example thereof [191]. The task was to implement a grammar for the (topological) Sierpinski triangle construction, where nodes play the role of vertices and edges serve to connect vertices to triangles. The corresponding relational growth grammar was already described in Ex. 5.23 on page 113. An implementation using the simple implementation of the XL interfaces can be easily done. At first, we have to declare a concrete subclass of `Node` for the vertices, this class also has to store the adjacency information. As the relational growth grammar uses three edge types $\mathtt{e}_0$, $\mathtt{e}_{120}$, $\mathtt{e}_{240}$, of which each vertex has at most a single outgoing edge, we can represent outgoing edges by providing a pointer for each edge type:

```
class Vertex extends Node {
    Node v0;
    Node v120;
    Node v240;
    ... // implementation of methods specified by Node
}
```

Then traversing edges in their direction is the fast operation of following a pointer, while traversing edges in opposite direction requires (in general) scanning the whole graph. If we are interested in a fast bidirectional traversability, we would have to provide also pointers for incoming edges, but for the Sierpinski grammar, this is not necessary.

The edge types themselves are declared by bit mask constants (Sect. 9.1.1 on page 235):

```
const int e0 = MIN_USER_EDGE;
const int e120 = MIN_USER_EDGE << 1;
const int e240 = MIN_USER_EDGE << 2;
```

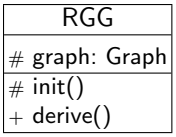
They are used for the `bits`-parameters of the methods of `Node`. As an example of the implementation of these methods, consider the method `addEdgeBitsTo`:

```
public void addEdgeBitsTo (Node target, int bits) {
    if ((bits & e0) != 0) {
        v0 = target;
    }
    if ((bits & e120) != 0) {
        v120 = target;
    }
    if ((bits & e240) != 0) {
        v240 = target;
    }
}
```

Now the single rule of the Sierpinski grammar (see again Ex. 5.23 on page 113) can be written as

```
a:Vertex -e0-> b:Vertex -e120-> c:Vertex -e240-> a ==>>
    a -e0-> ab:Vertex -e0-> b -e120-> bc:Vertex -e120-> c
                              -e240-> ca:Vertex -e240-> a,
    ca -e0-> bc -e240-> ab -e120-> ca;
```

However, we have to embed this rule in a complete program of the XL programming language. For this purpose, the simple implementation contains a class `RGG` which can be used as a superclass of a relational growth grammar and provides the necessary integration within the run-time system of the XL programming language:



This class has a constructor with a single **boolean** parameter. It creates a new instance of `Graph`, where the parameter tells the new graph whether the method `getAdjacentNodes` of nodes provides a bidirectional traversability or not. Then, the new graph is set up as the current graph (see Sect. 6.5.1 on page 142), the method `init` is invoked, and finally the method `derive`. By

default, `init` does nothing, but `derive` delegates to the method `derive` of the graph. This method performs the derivation using the parallel production which has been collected by previous rule executions (Sect. 9.1.2 on page 237). Using the class `RGG`, a complete program for the Sierpinski grammar is

```
import de.grogra.xl.impl.simple.*;

@de.grogra.xl.query.UseModel(CompiletimeModel.class)
public class Sierpinski extends RGG {
    const int e0 = RuntimeModel.MIN_USER_EDGE;
    const int e120 = RuntimeModel.MIN_USER_EDGE << 1;
    const int e240 = RuntimeModel.MIN_USER_EDGE << 2;

    static class Vertex extends Node {
        Node v0;
        Node v120;
        Node v240;
        ... // implementation of methods, see above
    }

    Sierpinski() {
        super(false);
    }

    protected void init() [
        ==>> a:Vertex -e0-> Vertex -e120-> Vertex -e240-> a;
    ]

    void rule() [
        a:Vertex -e0-> b:Vertex -e120-> c:Vertex -e240-> a ==>>
            a -e0-> ab:Vertex -e0-> b -e120-> bc:Vertex -e120-> c
                                      -e240-> ca:Vertex -e240-> a,
            ca -e0-> bc -e240-> ab -e120-> ca;
    ]

    void run(int n) {
        for (int i : 1:n) {
            rule();
            derive();
            System.out.println("Step " + i + ": " + graph.size()
                               + " nodes.");
        }
    }

    public static void main(String[] args) {
        new Sierpinski().run(Integer.parseInt(args[0]));
    }
}
```

We will use this grammar for a benchmark in Sect. 10.8.3 on page 359.

9.3 Document Object Model Implementation

A very popular representation of hierarchical structures is the textual XML data format (extensible markup language [15]). Besides some meta-information in the header, an XML document consists of a tree of elements, represented by textual tags, that may have attributes and contain other elements. An example is the document (from [85])

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN"
  "http://www.web3d.org/specifications/x3d-3.0.dtd">
<X3D version="3.0" profile="Interchange">
  <head>
    <meta name="filename" content="RedSphereBlueBox.x3d"/>
  </head>
  <Scene>
    <Transform>
      <DirectionalLight/>
      <Transform translation="3.0 0.0 1.0">
        <Shape>
          <Sphere radius="2.3"/>
          <Appearance>
            <Material diffuseColor="1.0 0.0 0.0"/>
          </Appearance>
        </Shape>
      </Transform>
      <Transform translation="-2.4 0.2 1.0" rotation="0.0 0.707 0.707 0.9">
        <Shape>
          <Box/>
          <Appearance>
            <Material diffuseColor="0.0 0.0 1.0"/>
          </Appearance>
        </Shape>
      </Transform>
    </Transform>
  </Scene>
</X3D>
```

It conforms to the X3D specification [84, 85] and, as such, represents 3D content primarily meant for interactive web applications.

As an XML document represents a tree of elements, an obvious memory-internal representation is as a tree of objects. The Document Object Model (DOM) standard [3] specifies such a representation, an implementation thereof for the Java programming language is contained in the package `org.w3c.dom` and subpackages of `javax.xml`. As this is a standard of the Java run-time environment, being able to let rules of the XL programming language operate on DOM trees would open the door for various applications where XML documents shall be inspected or transformed. Therefore, we created a DOM implementation of the XL interfaces on top of the base implementation. It uses

instances of `org.w3c.dom.Node` as nodes. XML does not specify analogues of edge types, but we can identify three relations: the relation of a node being a child of another node, the relation of a node being the next sibling of another node (note that children are ordered), and the relation of a node being an attribute node of an element node. For the latter two relations, edge types `SIBLING` and `ATTRIBUTE` are defined, while the child relation is modelled by the existence of a branch or a successor edge. I. e., these two edges types are considered equivalent within the DOM implementation.

The implementation is contained in the package `de.grogra.xl.impl.dom`. Its structure is the same as for the simple implementation, i. e., the class `Graph` implements the graph model, but now for DOM trees, and the class `RGG` provides an easy integration within the run-time system of the XL programming language. The DOM implementation also contains an implementation of properties: in XML documents each element node (an instance of `org.w3c.dom.Element`) may have attributes, these can be accessed as properties. The DOM implementation allows any possible name to be used, although XML documents typically restrict the set of attributes for an element by a document type definition or an XML schema. Attribute values are exposed as instances of `de.grogra.xl.util.Variant` by the DOM implementation. Variants can hold boolean values, numbers or strings and declare overloads for common operators so that the user can work with them in a convenient way without having to tell the compiler the exact type of an attribute. A more sophisticated DOM implementation could use a document type definition or a schema to determine the allowed attributes and their types already at compile-time.

The class `RGG` declares some auxiliary methods. The method `execute` parses its parameters into the names of an input and an output file and a number n of steps to be applied. The input file is then read as an XML file into the graph, the method `init` is invoked, n derivations using the method `run` are performed, the method `finish` is invoked, and the final result is written to the output file. To facilitate the usage of the DOM implementation, there are two auxiliary methods with the name `E` in class `RGG`:

```
public static boolean E(Element node, String name) {
    return (node.getNodeType() == Node.ELEMENT_NODE)
        && node.getTagName().equals(name);
}

public Element E(String name) {
    return document.createElement(name);
}
```

The first method can be used as a unary predicate (Sect. 6.5.2 on page 146) to test whether a given `node` is an element tagged with `name`. The test for elements is necessary as XML documents contain not only element nodes, but also nodes for the whole document, for text, attributes etc. The second factory method creates an element node tagged with `name`. E. g., the rule

```
E("big") ==> E("small");
```

transforms the part

```
<big>This is some text.</big>
```

of an XHTML document, which is represented in the DOM tree as a big-tagged element node with a single text node as child, to the part

```
<small>This is some text.</small>
```

Note that the rules for resolution of ambiguities between overloaded methods prefer methods with more arguments so that the left-hand side in fact uses the predicate method with two arguments, although the factory method is applicable, too.

9.3.1 Simple Model of Young Maple Trees

Now, let us use the DOM implementation for a first simple plant model of young maple trees. To have a representation of geometry, the X3D standard [84] is a natural choice: this already defines an interpretation of an XML document as a 3D scene graph, it allows the declaration and use of prototypes (comparable to classes in object-oriented programming), and it can be displayed in a lot of freely available browsers. For our model, we need five types of nodes: `Leaf` represents a leaf object (a rectangle with a leaf image as texture), `F` an internode (a cylinder along the local y-axis, for the botanical meaning see Sect. 5.2.1 on page 95 and [67]), and `RL`, `RU`, `RH` shall stand for rotations about the local x-, z- or y-axis, respectively. The names for the last four types are adopted from the GROGRA software (Sect. 3.15.1 on page 35, note that the role of the y- and z-axis is interchanged in X3D compared to GROGRA, the y-axis points upwards). Leaf objects always look the same, while the other types have geometric attributes (angle for rotations, length and radius for cylinders). To represent these entities of the model as nodes of an X3D document, we could therefore use a fixed master node (which can be multiply instantiated, see also Sect. 6.11.2 on page 182) for leaves and declare suitable prototypes for other types. The master node for leaves uses a very flat Box as geometry:

```
<Transform DEF="Leaf" translation="0 0.075 0">
  <Shape>
    <Box size="0.15 0.15 0.0001"/>
    <Appearance>
      <ImageTexture url="MapleLeaf.png"/>
    </Appearance>
  </Shape>
</Transform>
```

To instantiate such a leaf, one has to reference the master node as in

```
<Transform USE="Leaf"/>
```

But for our model, we need two additional non-geometric attributes for leaves. `order` shall stand for the branching order of the leaf (which is zero for the main stem, one for main branches etc.), `vitality` for the vitality which in our model shall be fixed by the topological position of the leaf. Furthermore, as the leaf type shall be used as pattern on left-hand sides and in queries, it is more convenient to represent the type directly by the tag name of the leaf node as in

```
<Leaf order="0" vitality="2"/>
```

This is no valid X3D node, but we can easily transform our model-specific nodes to X3D nodes in a final step by the XL rule

```
E("Leaf") ==> E("Transform").($[USE] = "Leaf");
```

The right-hand side makes use of a with-instance expression list (Sect. 6.13.4 on page 187) in which the special identifier `$` addresses the result of the expression `E("Transform")`. We use this to set the `USE`-attribute of the Transform-node to `"Leaf"`. If we declare a special predicate method for leaves in our model

```
static boolean Leaf(Element node) {
    return E(node, "Leaf");
}
```

we can write the rule in a more convenient way:

```
Leaf ==> E("Transform").($[USE] = "Leaf");
```

The representation of `F`-nodes is more complicated. The turtle command `F` draws a cylinder of the given length and radius and then translates the local coordinate system along the axis of the cylinder so that consecutive elements are stacked on top of each other. But X3D does not define such a node. Transform nodes can translate the coordinate system, but have no geometry, Cylinder nodes have geometry, but do not translate the coordinate system. Therefore, we have to declare a prototype, which in case of a fixed length 1 could look like

```
<ProtoDeclare name="F">
  <ProtoInterface>
    <field name="radius" type="SFFloat" accessType="initializeOnly"/>
    <field name="children" type="MFNode" accessType="initializeOnly"/>
  </ProtoInterface>
  <ProtoBody>
    <Transform translation="0 1 0"> <!-- shift of coordinate system -->
      <Group>
        <IS> <!-- insert child nodes of F in shifted coordinate system -->
          <connect nodeField="children" protoField="children"/>
        </IS>
      </Group>
      <Transform translation="0 −0.5 0"> <!-- undo half of translation -->
```

```
        <Shape>
          <Cylinder height="1"> <!-- insert cylinder -->
            <IS><connect nodeField="radius" protoField="radius"/></IS>
          </Cylinder>
          <Appearance><Material diffuseColor="0.7 0.55 0.4"/></Appearance>
        </Shape>
      </Transform>
    </Transform>
  </ProtoBody>
</ProtoDeclare>
```

For a variable length, we have to insert the length as the y-component of the first translation and as the length of the cylinder, and we have to use half of the negated length as the y-component of the second translation. Unfortunately, we cannot specify this by a prototype declaration with a single field `length` of type SFFloat in the interface as X3D does not provide built-in means to convert an SFFloat value to an SFVec3f value or to compute the negated half of an SFFloat value. The simplest way which we found is to use a single field `scale` of type SFVec3f which contains the value $(1, d, 1)$, d being the length. If we enclose the translated cylinder of unit length by an additional Transform node with such a scaling, the cylinder has the desired size. Furthermore, we can use the scaling as a translation vector, too, if we prepend an additional translation by $(-1, 0, -1)$:

```
<ProtoDeclare name="F">
  <ProtoInterface>
    <field name="scale" type="SFVec3f" accessType="initializeOnly"/>
    <field name="radius" type="SFFloat" accessType="initializeOnly"/>
    <field name="children" type="MFNode" accessType="initializeOnly"/>
  </ProtoInterface>
  <ProtoBody>
    <Transform translation="-1 0 -1">
      <Transform>
        <IS><connect nodeField="translation" protoField="scale"/></IS>
        <Group> ... </Group>
        <Transform>
          <IS><connect nodeField="scale" protoField="scale"/></IS>
          <Transform translation="0 -0.5 0"> ... </Transform>
        </Transform>
      </Transform>
    </Transform>
  </ProtoBody>
</ProtoDeclare>
```

An instance of this prototype is specified like

```
<ProtoInstance name="F">
  <fieldValue name="radius" value="0.04"/>
  <fieldValue name="scale" value="1 0.2 1"/>
  <fieldValue name="children">
```

```
    <!-- child nodes of F, e.g., <Transform USE="Leaf"/> -->
  </fieldValue>
</ProtoInstance>
```

Again, this representation is not feasible for direct use in our model. A representation like

```
<F radius="0.4" scale="1 0.2 1">
  <!-- child nodes -->
</F>
```

is more convenient and can be transformed to a correct X3D structure by a single rule:

```
f:F ==>
    E("ProtoInstance").($[name] = "F")
    for((* f -attr-> a:Attr *)) (
        [ E("fieldValue").($[name] = a.getName(),
                          $[value] = a.getValue()) ]
    )
    E("fieldValue").($[name] = "children");
```

This rule creates a ProtoInstance node with a fieldValue child for each attribute of the original node and a final fieldValue child to which all children of the original node are moved by the implicit connection transformations of the rule arrow `==>`. For the left-hand side, we again have to declare a predicate method for F-tagged elements.

The three rotation types `RL`, `RU`, `RH` could also be represented by prototypes, but as we will not need them on left-hand sides, this is not necessary. We can simply forget their original type and use standard Transform nodes with suitable rotations. This is most easily handled by factory methods like

```
Element RL(float angle) {
    return E("Transform").($[rotation] = "1 0 0 "
                                       + Math.toRadians(angle));
}
```

which have to be included in the code of our model, but could also be provided by some general library for XL-based modelling of X3D scenes. We also declare factory methods for `F` and `Leaf`:

```
Element F(float length, float radius) {
    return E("F").($[scale] = "1 " + length + " 1", $[radius] = radius);
}

Element Leaf(int o, float v) {
    return E("Leaf").($[order] = o, $[vitality] = v);
}
```

After having defined the types of our model, we start with the initialization of the model. This is done in the `init` method which is automatically invoked (see above):

```
protected void init() [
    s:E("Scene") ==>> s Leaf(0, 2);
]
```

This appends a new `Leaf` node to the single `Scene` node which has to be present (together with the prototype declaration for `F` and the definition of the master node for leaves) in the input XML document. This leaf shall now be the germ out of which the whole tree grows. I. e., in botanical terms, a `Leaf` node represents also a bud. The structure of a typical growth rule with opposite leaf arrangement, as it is the case for maple trees, is

```
Leaf ==> F [RU(a) Leaf] [RU(-a) Leaf] F Leaf;
```

with the branching angle `a`: this rule creates a first internode with two branches, which initially only consist of a leaf (which is at the same time a bud), on opposite sides of the growth axis, and then a second internode with a terminal leaf (and bud) that continues the current axis. Depending on plant species and growth conditions, there may also emerge more than one branch-bearing internode in a single growth step. For our model, we use a loop which creates two branch-bearing internodes. Furthermore, we take into account that the leaf arrangement of maple trees is opposite-decussate (adjacent pairs of opposite leaves are at right angles) and distinguish between growth of the main stem and growth of the branches. With a more or less guessed parameterization (i. e., the parameters do not stem from measurements, but only from manual optimization of the subjective impression of the 3D outcome), we arrive at this growth rule:

```
x:Leaf ==>
    {
        int o = x[order];
        float v = x[vitality]; // vitality of leaf/bud
        // reduce vitality for new branches by order-dependent factor
        float bv = v * ((o == 0) ? 0.8f : 0.77f);
        float len = 0.1f * (v + 1)**0.6f; // length of new internodes
    }
    // if not on main stem, let shoot emerge only with a vitality-
    // dependent probability
    if ((o == 0) || (rnd.nextFloat() < v - 0.5f)) (
        F(len, 0)
        for (int i : 1:2) ( // two branch-bearing internodes
            if (o == 0) (
                // main stem
                [RL(normal(60, 15)) Leaf(o+1, bv)]
                [RL(normal(-60, 15)) Leaf(o+1, bv)]
                // decussate growth: rotate by 90 degrees
                RH(normal(90, 10))
                F(len, 0)
            ) else (
                // branches
```

```
                [RU(normal(52, 11)) Leaf(o+1, bv)]
                [RU(-normal(52, 11)) Leaf(o+1, bv)]
                // some random angle
                RU(normal(0, 10))
                // damped decussate growth
                RH(normal((i == 1) ? 10 : -10, 10))
                F(len, 0)
            )
        )
        // terminal leaf, reduce vitality only within branches
        Leaf(o, (o == 0) ? v : v * 0.9f)
    );
```

In this rule, the radii of internodes were initialized with 0. To compute their actual values, we use a variant of the pipe model [37]. In its simplest form, this model has already been stated by Leonardo da Vinci and says that the sum of the cross sections behind a branching is the same as the single cross section before the branching. In our variant, we compute in a recursive post-order traversal for each internode `F` the sum of the cross sections of its direct internode children plus the sum of the cross sections of the leaves which the internode bears. The result is taken as the cross section of the internode, and a new radius is computed. Only if this exceeds the current one, we use this radius as the new radius of the internode. If the new radius even falls below half of the current radius, the supply with assimilates by leaves is assumed to be insufficient, and the internode falls off. This is a simple implementation of self-pruning of the tree.

```
static float computeCrossSection(Element node) {
    // sum up contribution of direct children
    float cs = sum(computeCrossSection((* node (-child->)+ : (F) *)))
    // and direct leaves (the condition !F(c) ensures that the closure
    // of the child relation does not pass internodes). A fixed cross
    // section of 5 mm^2 is assigned to each leaf
        + 5e-6f * count((* node (-child-> c:Element & !F(c))+ Leaf *));
    // compute radius, add some amount
    float r = (float) Math.sqrt(cs / Math.PI) + 0.004f;
    if (r > node[radius]) {
        node[radius] = r; // secondary growth: increase radius
    }
    else if (r < 0.5f * node[radius]) [
        node ==>> ; // self-pruning: fall off due to insufficient supply
    ]
    return cs;
}
```

The above growth rule has to be combined with the computation of cross sections to a single step of the model in the `run` method. A derivation using the parallel production which results from the application of the growth rule has to be performed before cross sections are updated. The derivation is triggered

by the method `derive` (Sect. 9.1.2 on page 237), so we have to invoke this method before `computeCrossSection`:

```
protected void run() {
    [
        x:Leaf ==> ... ; // growth rule, see above
    ]
    derive(); // perform derivation, so results are visible from now on
    computeCrossSection((* ^ (-child->)+ : (F) *));
}
```

The argument of `computeCrossSection` is a query which starts at the root node (symbol `^`) and finds the first internode. The final two rules for `Leaf` and `F` which transform the model representation to a true X3D document are specified as the body of the `finish` method:

```
protected void finish() [
    f:F ==> ...;  // see above
    Leaf ==> ...; // see above
]
```

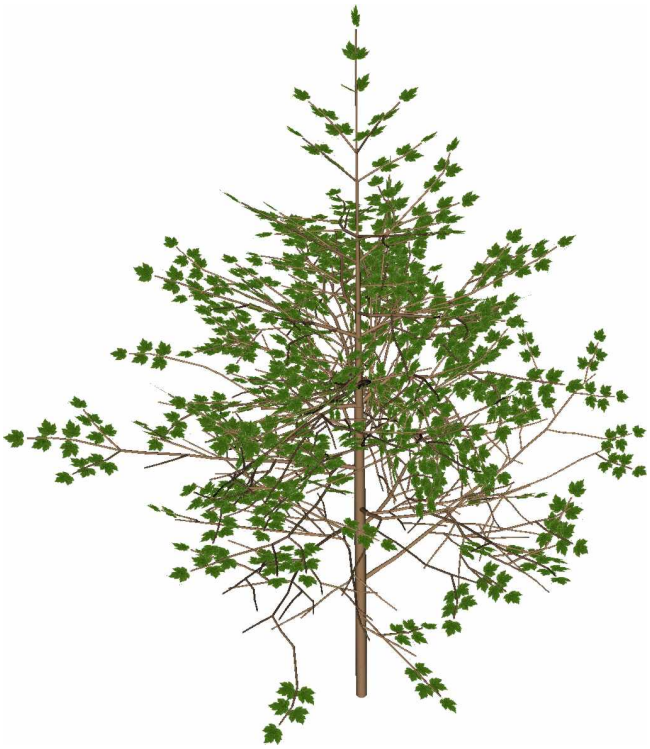


Figure 9.6. Outcome of the maple model after ten steps

Now if we apply this model to an initial XML document which contains an empty X3D scene and the definitions of the master node for leaves and the prototype for internodes, we obtain an X3D document as result which represents a virtual maple tree after having grown the specified number of steps. The outcome can be visualized within an X3D browser, Fig. 9.6 on the preceding page shows a result after ten steps.

The main advantage of the approach of using XL in combination with X3D for plant modelling is the utilization of international standards. We do not need some specialized L-system software or a complex proprietary 3D system, we only need the XL programming language in combination with its DOM implementation and an X3D browser. Already this system enables us to specify global interactions between the model entities like the self-pruning due to an insufficient number of leaves. On the other hand, the introduction of types of the model into X3D by prototypes and master nodes is relatively complicated, and the run-time performance is far from good. Also the inclusion of spatial sensitivity is not possible with this simple system as it is only the browser which equips the structure with a 3D interpretation. We would have to provide a set of X3D-aware methods to compute, within the DOM tree, properties like the global position of nodes in 3D space or their distance. Furthermore, for a convenient modelling workflow we would need an integrated development environment.

For XML documents, there also exist standards to query information and to transform a document, namely XPath [25] and XQuery [12] for queries and XSLT [24] for transformations. We have not yet made a thorough comparison of these standards with the query and transformation possibilities of the DOM implementation, but this is clearly an interesting topic. For example, the two model transformations in the method `finish` of the maple tree model would be typical applications of XSLT if we had implemented them using XSLT.

9.4 Implementation for Commercial 3D Modellers

The flexibility of the graph interface of the XL programming language allows an implementation for any given custom graph model. A very attractive application thereof is the implementation for existing 3D modellers. These represent 3D content as a scene graph, which is normally edited manually. Examples are CAD applications in engineering and architecture, visual effects in animated films, or even complete animated films. But of course, manual editing has its limits, and this is where automated content creation by graph grammars and other techniques comes into play.

The following three implementations were done by students as part of their bachelor theses. As the native programming language of the used modellers is C++, the Java Native Interface [185] had to be used to link their scene graph with the XL programming language.

9.4.1 CINEMA 4D

René Herzog was the first to implement the XL interfaces for a commercial 3D modeller, namely for CINEMA 4D [78, 126]. The structure of the scene graph turned out to be suitable for rule-based modelling, in particular it was possible to implement several L-systems in the same way as for dedicated L-system software. We do not give the details as they can be found in the thesis. Figure 9.7 on the facing page shows some figures which were created with the developed XL4C4D plug-in. Unfortunately, the XL interfaces have been considerably changed since then, but an adaptation should be possible with reasonable effort.

9.4.2 3ds Max

Uwe Mannl implemented a plug-in for the 3D modeller 3ds Max [125, 4]. As for CINEMA 4D, the scene graph structure allowed a direct translation from L-system symbols to scene graph nodes. Figure 9.8 on the next page shows two models taken from the thesis.

9.4.3 Maya

Udo Bischof wrote a plug-in for the 3D modeller Maya [10, 5]. This modeller has a very sophisticated scene graph where even simple geometric objects like spheres are represented by several cooperating nodes with connecting dataflow edges. While this is a very powerful technique in principle, it hindered a straightforward implementation of the XL interfaces: if Maya nodes and XL nodes were in a one-to-one correspondence, the user would have to specify several nodes in XL rules to address a single entity in terms of the model. Thus, the implementation had to simplify the graph structure which is visible through the XL interfaces so that the user only sees a structure similar to axial trees of L-system words and suitable for rule-based modelling.

Besides providing the basic functionality, Udo Bischof also implemented several useful features for global 3D sensitivity. E. g., it is possible to query information about the closeness to a given geometric structure. This was used in the models shown in Fig. 9.9 on page 268: the city generator uses a manually modelled landscape and (invisible) bounding objects to govern the “growth” of the city, the second example lets plants creep along a predefined head-shaped surface.

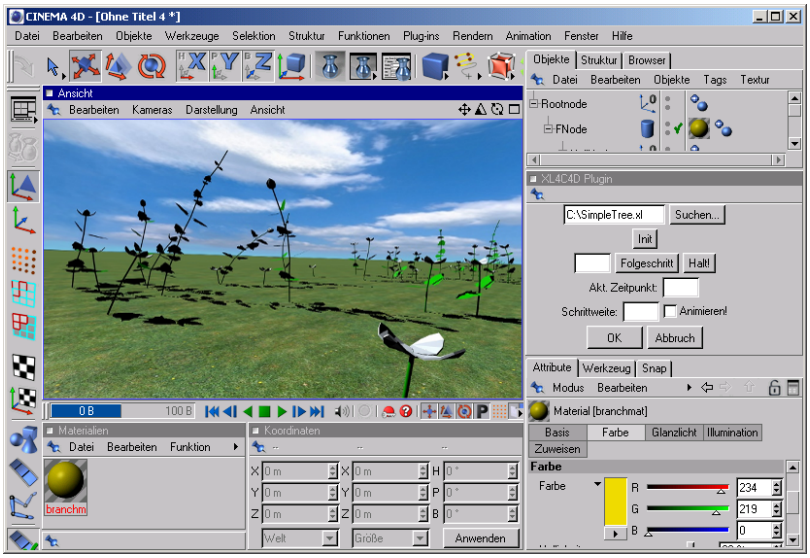


Figure 9.7. CINEMA 4D window displaying a field of flowers

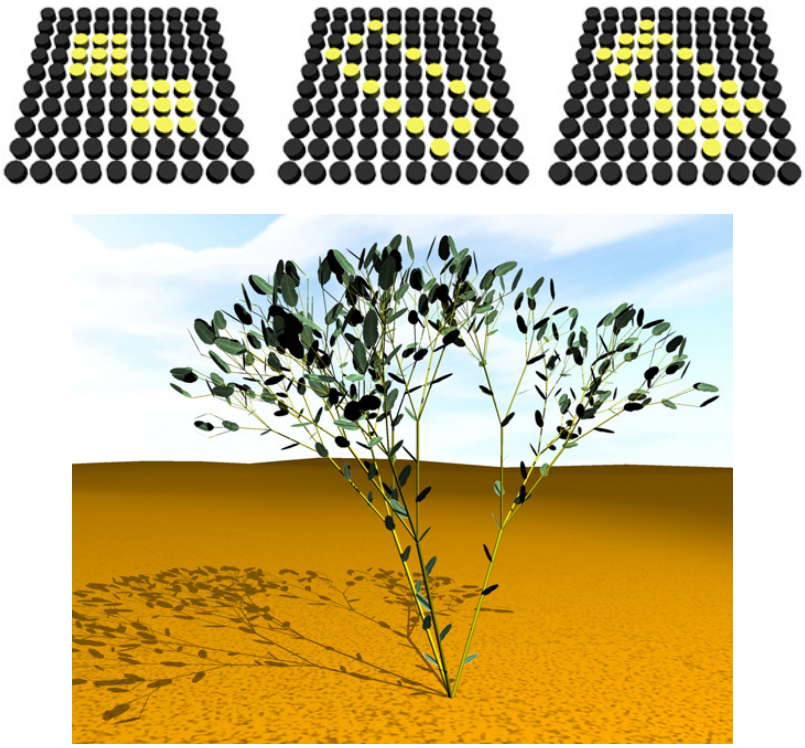


Figure 9.8. Game of Life and simple bush made with the XL plug-in for 3ds Max

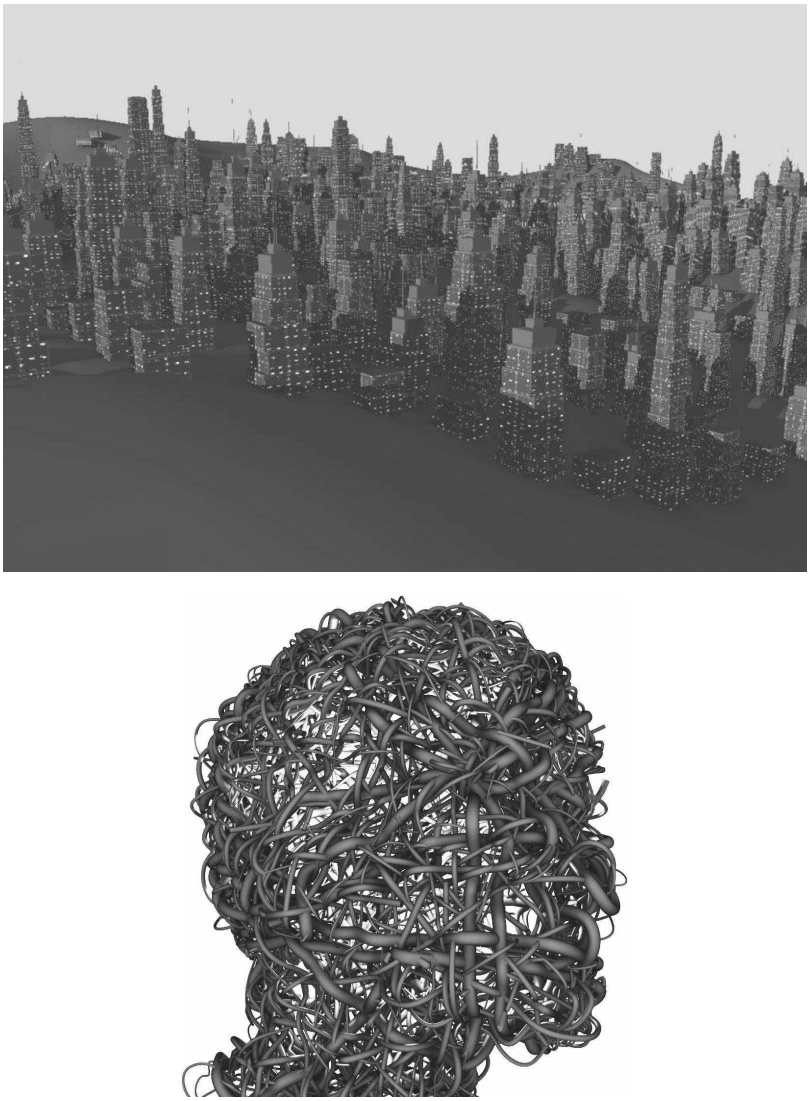


Figure 9.9. Using XL within Maya: city generator on a curved landscape and creeping plants on a head-shaped surface

10

Applications within GroIMP

In this chapter, we present a diverse collection of applications of the XL programming language within the modelling platform GroIMP (see Appendix A). The RGG plug-in of this platform contains the most sophisticated currently existing implementation of the XL interfaces and is especially tailored for the needs of 3D modelling, in particular virtual plant modelling. Most of the examples of this chapter are available in the example gallery which is contained in GroIMP distributions [101]. And most of the examples make use of the simplified RGG dialect of the XL programming language (Sect. B.4).

The examples often refer to classes and methods of GroIMP and the RGG plug-in. More detailed explanations can be found in Appendix A and B and in the API documentation [101].

10.1 Introductory Examples

This short section shows three simple examples. They are the implementation of examples from the introduction (Chap. 2) and illustrate the basic use of the XL programming language within GroIMP.

10.1.1 Snowflake Curve

The snowflake curve was presented in Sect. 2.1 on page 11 as a classical example for the rule-based paradigm. Its implementation using L-systems was shown in Sect. 3.2 on page 19 and, for the concrete syntax of GROGRA, in Sect. 3.15.1 on page 35. But these two implementations did not take into account the size reduction to one third. This is improved in our implementation using the XL programming language within GroIMP. If we use the simplified RGG dialect, the complete source code is:

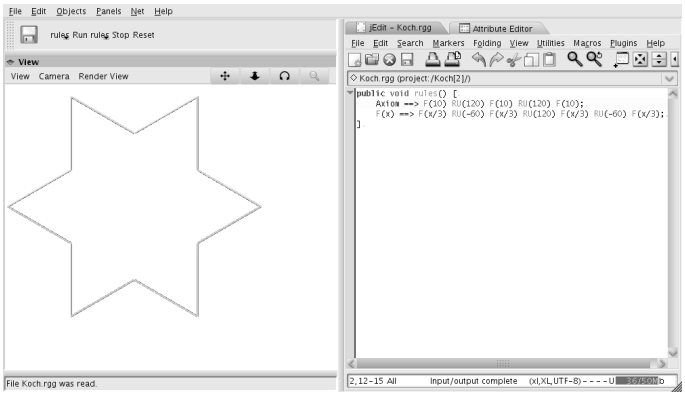
```
public void rules() [
    Axiom ==> F(10) RU(120) F(10) RU(120) F(10);
    F(x) ==> F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3);
]
```

As the method `rules` is parameterless and public, a button to invoke the method is created within the RGG toolbar of GroIMP (see Fig. 10.1 on the facing page). The body of the method consists of two rules with rule arrow `==>`, i. e., the RGG implementation automatically adds connection transformations so that the rules behave like L-system rules, but in a graph setting (Sect. 9.1.3 on page 239).

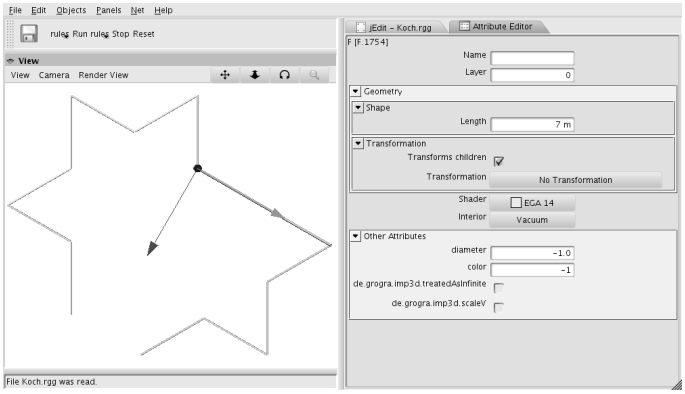
The first rule replaces nodes of class `Axiom` by a chain of `F` nodes with `RU` nodes between. The class `Axiom` is a member of the package `de.grogra.rgg` which is imported automatically when the RGG dialect is used. The initial graph contains exactly one such `Axiom` node if we use the class `RGG` as superclass for our model, which is again implicit for the RGG dialect. Thus, the first rule is applicable exactly once. The classes used on the right-hand side, `F` and `RU`, are members of the package `de.grogra.turtle` which is also imported automatically. This package contains a collection of turtle command classes, but now seen as nodes of a scene graph. Their names and parameters were chosen to conform with the GROGRA software (Sect. 3.15.1 on page 35, [103]). I. e., an instance of `RU` is a scene graph node without geometry, but which transforms the local coordinate system by a rotation about the local y-axis. An instance of `F` has a cylinder as geometry whose base point (centre of the base cap) coincides with the origin of the local coordinate system and which extends along the z-axis. Furthermore, the local coordinate system of children of `F` nodes is transformed along the axis of the cylinder such that the local origin for children coincides with the top point (centre of the top cap) of the cylinder. The node expression `RU(120)` refers to a constructor declared in the class `RU`, i. e., the expression **`new`** `RU(120)` is evaluated which creates a new `RU` node with an angle of 120 degrees. Likewise, `F(10)` refers to a constructor for `F` and creates a cylinder with a length of 10 (and a default diameter). As the right-hand side separates nodes by whitespace, its application creates a chain of scene graph nodes which are connected by successor edges (Sect. 9.1.4 on page 243). This chain represents an equilateral triangle with side length 10.

The second rule replaces nodes of class `F` by a representation of the generator shape of the Koch construction. It makes use of a user-defined pattern declared in `F` with the signature `(@In @Out F node,` **`float`** `length)`: this is a parameterized pattern which matches nodes of class `F` and associates their length with its parameter `length`. So in the rule, the query variable `x` is bound to the length of the matched `F`, and we use one third thereof for the length of the four `F` nodes of the successor graph of the right-hand side.

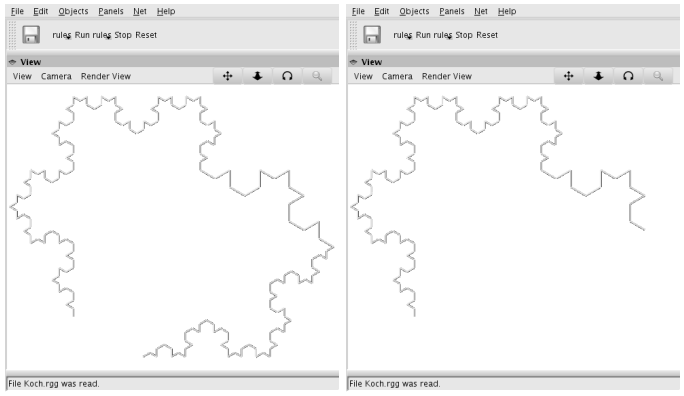
Now when the user clicks on the button labelled “rules”, the method `rules` is invoked once, and as an implicit final step the method `derive` is invoked on the graph. The latter triggers the parallel derivation using the previously executed rules, see Sect. 9.1.2 on page 237. The first derivation replaces the `Axiom` node by the initiator of the snowflake curve, further derivations apply the generator rule. The size of the graph grows quickly: after n applications of the generator, $3 \cdot 4^n$ `F` nodes and $3 \cdot 4^n - 1$ `RU` nodes have been created.



(a)



(b)



(c) (d)

Figure 10.1. GroIMP window showing the snowflake example: (**a**) after two steps; (**b**) one cylinder selected and its length modified; (**c**) after two further steps; (**d**) deletion of a node and its subgraph

Figure 10.1(a) on the previous page shows the GroIMP window with the RGG toolbar containing the "rules" button and the 3D visualization of the graph.

It is now possible to interact with the model: by clicking on a cylinder with the mouse, the corresponding `F` node becomes selected, and its attributes are shown in the attribute editor where they may also be modified. In general, it depends on the model whether such a modification is taken into account within further steps of the model or not. In this case, the model uses the current value of the length parameter as variable `x` in the generator rule. Thus, we may modify the length as shown in Fig. 10.1(b) and obtain the result that the part of the curve which emerges out of the modified `F` has a different size than the other parts even after applications of the generator rule. This is shown in Fig. 10.1(c). It is also possible to delete a selected node. This implicitly deletes all children of this node, too, or to be more precise, all nodes which are no longer connected with a distinguished root node of the graph after deletion of the selected node. An example thereof is shown in Fig. 10.1(d).

10.1.2 Sierpinski Triangles

The next example implements the rules of the Sierpinski triangle (Sect. 2.3 on page 13) in a graph setting. We use the variant described in Ex. 5.23 on page 113 and [191] which was also implemented using the simple implementation of the XL interfaces (Sect. 9.2.1 on page 253). The declaration of a class for vertices, of edge types and the methods `init` and `rule` are similar to the latter implementation with the exception that we do not need to define a mechanism to store and query edge information as the graph of GroIMP manages edges itself. I. e., the whole model can be specified by:

```
module Vertex;

const int e0 = EDGE_0;
const int e120 = EDGE_1;
const int e240 = EDGE_2;

protected void init() [
    Axiom ==>> ^ a:Vertex -e0-> Vertex -e120-> Vertex -e240-> a;
]

public void rule() [
    a:Vertex -e0-> b:Vertex -e120-> c:Vertex -e240-> a ==>>
        a -e0-> ab:Vertex -e0-> b -e120-> bc:Vertex -e120-> c
                                  -e240-> ca:Vertex -e240-> a,
        ca -e0-> bc -e240-> ab -e120-> ca;
]
```

Here, the initiator rule is contained in the special method `init` which is invoked automatically by GroIMP as part of the initialization of a model. For

the three model-specific edge types, we use the edge types `EDGE_0`, `EDGE_1`, `EDGE_2` which have no specific meaning within GroIMP. These edge types are declared in the library class `de.grogra.rgg.Library`, as static members they are automatically imported in the RGG dialect.

In the simple version, there is no visualization of the Sierpinski graph. For a 3D visualization, we have to compute suitable locations in 3D space for vertices, and vertices have to be represented by some geometry. To address the first issue, we use the class `de.grogra.imp3d.objects.Null` as superclass for vertices by the declaration

```
module Vertex extends Null;
```

`Null` objects have no geometry, but store a transformation of the local coordinate system. In usual terms of scene graphs (e. g., see the X3D specification [84] and Sect. 9.3.1 on page 258), they represent transform groups. We set the positions of the initial three vertices right in the initiator:

```
Axiom ==>>
    ^   // append the three initial vertices as direct children
        // of the root node and place them in 3D space
        [a:Vertex.(setTransform(-1, 0, 0))]
        [b:Vertex.(setTransform(1, 0, 0))]
        [c:Vertex.(setTransform(0, Math.sqrt(3), 0))],
    // finally create the edges for the triangle
    a -e0-> b -e120-> c -e240-> a;
```

In order for this to work as desired, it is important how the graph of GroIMP is interpreted as a scene graph (see Sect. A.3.1 on page 379). The scene graph is taken to be the part of the original graph which can be reached by following successor or branch edges from the root, and this subgraph has to be a tree to avoid ambiguities for derived attributes like the local coordinate system. I. e., the three model-specific edge types of our example are irrelevant for 3D visualization, and we have to create additional edges. This is done in the first lines of the rule which connects all vertices directly with the root, which is represented by the symbol `^`. In such a flat hierarchy, the coordinate transformations of the vertices (namely translations) do not interfere with each other.

Now the generator rule has to be modified in a similar way. To compute the new positions, we make use of operator overloading and user-defined conversions. The class `de.grogra.vecmath.VecmathOperators` contains operator overloads to compute the sum of 3D points (represented by `javax.vecmath.Point3d`) and to divide 3D points by a scalar value, and the class `de.grogra.rgg.Library` contains a conversion method which converts nodes into their 3D locations. Thus, for two nodes `a`, `b` the expression `(a+b)/2` computes the centre of these nodes in 3D space. We then write the rule as follows:

```
a:Vertex -e0-> b:Vertex -e120-> c:Vertex -e240-> a ==>>
```

```
    ^   // create the three new vertices as direct children of the
        // root node, compute their 3D position
        [ab:LLVertex.(setTransform((a+b)/2))]
        [bc:Vertex.(setTransform((b+c)/2))]
        [ca:LLVertex.(setTransform((c+a)/2))],
    // create the new edges for the triangles
    a -e0-> ab -e0-> b -e120-> bc -e120-> c -e240-> ca -e240-> a,
    ca -e0-> bc -e240-> ab -e120-> ca;
```

As a result, vertices are placed at their correct location. However, they still do not have geometry. One possibility would be to let `Vertex` extend from classes with geometry like `de.grogra.imp3d.objects.Sphere` instead of `Null`, but then we would have to shrink the size of the geometry of each vertex in each step. We use another possibility: we declare a single sphere as a member of the RGG class:

```
Sphere sphere = new Sphere().(setShader(YELLOW));
```

The method `setShader` is used to set the visual appearance of the sphere. Here, we set it to the plain colour `YELLOW` which is declared in the class `de.grogra.imp3d.shading.RGBAShader`. (Note that static members of this class are automatically imported by the RGG dialect.) The single sphere is shrunken in each step:

```
public void rule() {
    [
        ... // rule as above
    ]
    // shrink sphere
    sphere[radius] *= 0.5;
}
```

This uses a property access `sphere[radius]` to modify the property `radius` of the sphere, see Sect. 6.10 on page 174. Now we use the technique of object instancing (see Sect. 6.11.2 on page 182) to represent each vertex by the geometry of this single sphere:

```
module Vertex extends Null ==> INSTANCE.sphere;
```

Note the usage of the special field `INSTANCE` which is automatically declared in each RGG class by the RGG dialect. As we declared `sphere` as an instance member (i. e., not as a static member) of the RGG class, we can only address `sphere` in the context of an instance of the RGG class. Usually, RGG classes are singletons, i. e., there exists only a single instance of such a class within a single GroIMP project. This single instance can be accessed by the static field `INSTANCE`.

If we also want to visualize the edges of the Sierpinski triangle graph, we can again make use of instantiation rules. Now, for each outgoing edge of a vertex we additionally draw a thin white cylinder along the edge:

```
module Vertex extends Null ==>
    INSTANCE.sphere
    for ((* this -(e0|e120|e240)-> t:Vertex *)) (
        [
            Cylinder(0, 0.4 * INSTANCE.sphere[radius])
                .(setEndPoints(ORIGIN, t - this),
                  setShader(WHITE))
        ]
    );
```

The computation of the base and top points of the cylinder has to be relative to the vertex. I. e., the base point is the local origin $(0, 0, 0)$, for which the constant `ORIGIN` defined in `Library` can be used, and the top point is given by the difference vector of the locations of the two adjacent vertices. For this computation, we may simply write `t - this` as `Library` defines a user-defined conversion from nodes to positions and `de.grogra.vecmath.VecmathOperators` a suitable overload of the subtraction operator. The result after 4 steps is shown in Fig. 10.2.

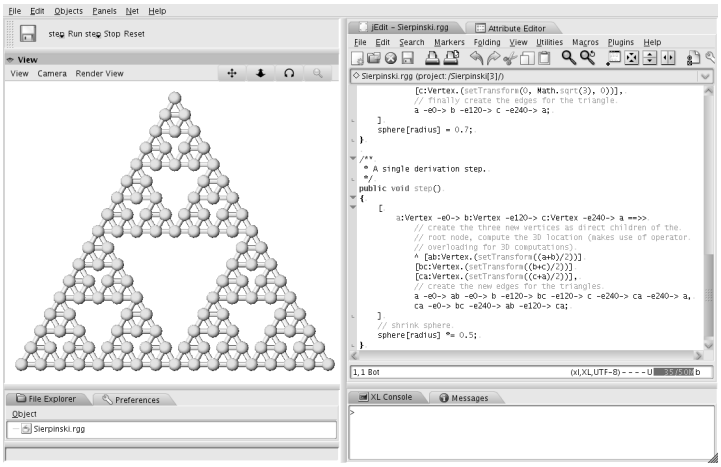


Figure 10.2. GroIMP window showing the Sierpinski example with edges

10.1.3 Game of Life

The Game of Life (Sect. 2.4 on page 14) can be implemented easily using XL and GroIMP [96, 109]. The package `de.grogra.rgg` contains the class `Cell` which has an **int**-valued attribute named `state` and is drawn as a cylinder with a `state`-dependent colour and radius. The initial world of the cellular automaton is created in the `init`-method as a square grid of 10 by 10 cells, where we assume `initialState` to be an **int**-valued method which defines the

initial state at cell (x, y). For the Game of Life, we use the value 0 for dead cells and 1 for living cells.

```
protected void init() [
    Axiom ==>> ^
        for (int x : 1:10) for (int y : 1:10) (
            [Cell(x, y, 0, initialState(x, y))]
        );
]
```

The graph structure does not yet reflect the topology of the grid. For the Game of Life, we need the Moore neighbourhood of all eight surrounding cells, see Fig. 10.3. This can be defined in geometric terms if we use the maximum norm L_∞ where the distance between two points is the maximum of the distances in each dimension, i. e., $d_\infty(a, b) = \min_{i=0\ldots2} |b_i - a_i|$ for three dimensions. The class `Cell` provides the method `distanceLinf` to compute this distance so that we can define a neighbourhood-relation

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

We use 1.1 instead of 1 to account for numerical imprecision.

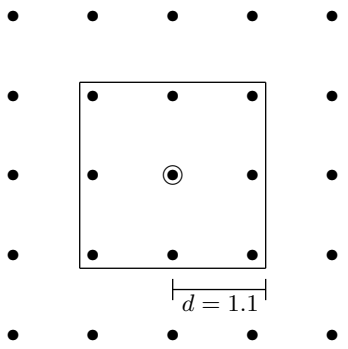


Figure 10.3. Moore neighbourhood

Using this relation, we can compute the number of living neighbours of a cell `x` by a combination of a query with the aggregate method `sum` declared in the automatically imported class `de.grogra.xl.util.Operators` (Sect. 6.4.3 on page 140): `sum((* x -neighbour-> Cell *)[state])`. This expression is used in the two transition rules of the Game of Life:

```
public void transition() [
    x:Cell(1), (!(sum((* x -neighbour-> Cell *)[state]) in (2 : 3)))
        ::> x[state] := 0; // dead due to loneliness or overcrowding
```

```
    x:Cell(0), (sum((* x -neighbour-> Cell *)[state]) == 3)
        ::> x[state] := 1; // cell comes to life
]
```

The class `Cell` also declares an operator overload for invocation which takes a single **int**-argument. It implements a shortcut for the assignment of a new `state` value, i. e., we could also write `x(0)` instead of `x[state] := 0`. Figure 10.4 shows a part of the famous glider sequence.

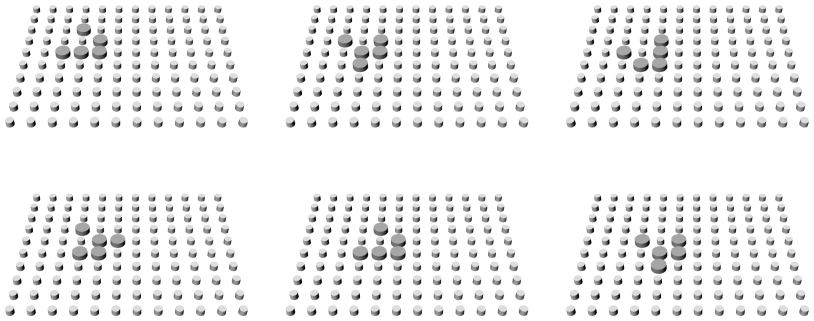


Figure 10.4. Game of Life with the glider pattern. After four steps, the pattern reproduces itself, but displaced diagonally in the grid

The geometric definition of the neighbourhood has two advantages: it is simple to specify, and if we do not arrange cells in a regular lattice, we can study the effects of lattice defects and dislocations. On the other hand, the geometric definition leads to an inherently inefficient implementation as within each transition and for each cell, the whole graph has to be scanned for neighbours. Given a grid size of $n \times n$, this amounts to a complexity of $\mathcal{O}(n^4)$ for a transition. An $\mathcal{O}(n^2)$-implementation can be obtained if the neighbourhood-relation is "materialized" by edges in the graph. Most easily, we implement this on top of the geometric definition, i. e., in a single preprocessing step (with complexity $\mathcal{O}(n^4)$) we create edges for nodes which are in geometric neighbourhood:

```
const int nb = EDGE_0;

protected void init() [
    Axiom ==>> ...;  // rule from above
    {derive();}
    c1:Cell, c2:Cell, (neighbour(c1, c2)) ==>> c1 -nb-> c2;
]
```

Then the number of living neighbours is computed by the $\mathcal{O}(1)$-operation `sum((* x -nb-> Cell *)[state])`. Note the invocation of `derive`: this triggers the derivation which actually inserts the initial grid into the graph, see

Sect. 9.1.2 on page 237. Without this invocation, the following rule would not see the grid, but still only the initial `Axiom`.

The Game of Life example shows that, within relational growth grammars and the XL programming language, we can easily and naturally represent and use arbitrary relations between entities. As the variant with a geometric neighbourhood relation does not make use of the graph structure, an implementation by an L-system would be possible, too, if the used software supports a query for the neighbours. An L-system implementation of the variant with explicit neighbourhood edges, if possible at all, would be very intricate as the two-dimensional structure of the grid has to be somehow encoded in the one-dimensional structure of the L-system word.

10.2 Technical Examples

The two examples of this section are of abstract, technical nature and shall illustrate the supported derivation modes and the possibility to nest rules.

10.2.1 Derivation Modes

The aim of this example is the illustration of the different derivation modes of the base implementation presented in Sect. 9.1.5 on page 245. The basic modes of sequential or parallel derivations exist in deterministic and nondeterministic variants, and the deterministic parallel mode can also be modified so that a node is deleted at most once. The example consists of the following code:

```
module A;
module B;
module C;
module D;

protected void init() {
    [
        Axiom ==>> ^ -EDGE_0-> A for (1:4) (A);
    ]
    derive();
    setDerivationMode(PARALLEL_MODE | EXCLUDE_DELETED_FLAG);
    for (applyUntilFinished()) {
        println(graph().toXLString(false));
        [
            A ==> B;
            A ==> C;
            A ==> D;
        ]
    }
}
```

The initialization creates a linear chain of five `A` nodes connected by successor edges. An edge of the general-purpose type `EDGE_0` connects the head of this chain with the root node (symbol `^`). Thus, the `A` nodes are not reachable from the root by branch or successor edges, and the restriction for these edges types to span only a tree (Sect. 10.1.2 on page 273) does not apply. The invocation of `derive` ensures that the effects of the initialization rule are applied to the graph. Then we set the derivation mode to the default `PARALLEL_MODE | EXCLUDE_DELETED_FLAG`, i. e., all rules shall be applied in parallel, but if the execution of the right-hand side of a rule leads to a node deletion entry in the queue collection, then the corresponding node is excluded from further matches. This induces a rule priority given by the control flow, i. e., rules which are executed at first have a higher priority. This is common practice in L-system software and allows to list alternatives as in

```
A(x) & (x > 1) ==> ...;
A(x) & (x > 0) ==> ...;
A(x)           ==> ...;
```

The first applicable rule then replaces the `A` node, the following are not taken into account.

In the example, we use the method `applyUntilFinished` to execute three rules as long as possible, i. e., as long as there is a change to the graph. As the rule `A ==> B;` is executed at first, it has the highest priority for the chosen derivation mode. Thus, all `A` nodes are replaced by `B` nodes in a single step. This can be seen by the textual output of the `println`-statement which uses the method `toXLString` to represent the graph as a string in the syntax of right-hand sides. After removing surrounding constant text for the root node and the `EDGE_0` edge, the output is

```
A A A A A
B B B B B
```

Changing the derivation mode to `SEQUENTIAL_MODE`, we obtain a different output:

```
A A A A A
B A A A A
B B A A A
B B B A A
B B B B A
B B B B B
```

For a single derivation step, the sequential mode only uses the first match of all matches of all rules, thus the rule `A ==> B;` has highest priority again. That the nodes are processed from left to right in the textual output is a result of the order in which the nodes were added in the initialization: for two nodes of the same class, the older node is found at first.

The derivation mode `SEQUENTIAL_NON_DETERMINISTIC_MODE` chooses one match out of all possible matches. A possible derivation sequence is the following:

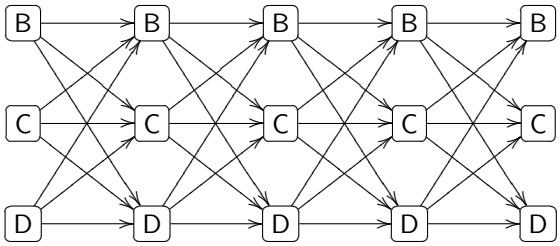
```
A A A A A
C A A A A
C A D A A
C B D A A
C B D C A
C B D C B
```

But the actual sequence of course depends on the used pseudorandom numbers.

The derivation mode `PARALLEL_NON_DETERMINISTIC_MODE` is based on all possible matches. For a pair of matches with overlapping sets of nodes to delete, it pseudorandomly discards one of the matches. A possible derivation is

```
A A A A A
D C D B D
```

The most complex result is obtained by the mode `PARALLEL_MODE` without `EXCLUDE_DELETED_FLAG`. This applies each rule to each match in parallel. In the example, each `A` node is replaced by a `B` node, a `C` node and a `D` node in parallel. The result is then no longer a chain of nodes, but a true graph with successor edges as connections:



The internal mechanism of the different derivation modes was explained in Sect. 9.1.5 on page 245. The crucial point for the two nondeterministic modes is that it is not possible to pseudorandomly choose the rule and match to apply in advance as there is no knowledge about all available rules. The run-time system rather executes all rules and finds all matches. In the parallel mode, for each match the corresponding right-hand side is also executed. In the sequential mode only some right-hand sides are executed (see Sect. 9.1.5 on page 245). But not any execution of a right-hand side leads to a change in the graph. At first, such an execution only leads to some entries in the modification queues. The run-time system then processes not all of these entries, but only those belonging to a subset of executed right-hand sides, where the chosen subset depends on the derivation mode. In order to see which right-hand sides are executed, we add textual output as follows:

```
A ==> B print("b ");
A ==> C print("c ");
A ==> D print("d ");
```

Then the sequence of nondeterministic sequential derivations yields the following output, where we can see that if more than one right-hand side is executed in a single step, only the last execution is the one whose modification entries in the queues are actually processed:

```
A A A A A
b c
C A A A A
b b b c c c d
C A D A A
b
C B D A A
b b c
C B D C A
b
C B D C B
```

In the nondeterministic parallel mode, the right-hand sides are executed for each match:

```
A A A A A
b b b b b c c c c c d d d d d
D C D B D
```

We obtain a more interesting example for the nondeterministic parallel mode if the left-hand sides are of different size so that the overlap of matches may be partial:

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

Starting with a sequence of ten `A` nodes, the results of four different derivations are

```
C C C D D D D B B A
A C C C A C C C B B
D D D D A C C C B B
B B D D D D B B B B
```

Thus, the run-time system has pseudorandomly chosen nonoverlapping matches out of all found matches. This may lead to parts not covered by matches as we can see from the remaining `A` nodes.

10.2.2 Amalgamated Two-Level Derivations

We presented amalgamated two-level derivations in Sect. 4.3.1 on page 64 and showed an example on page 65 which specifies an edNCE embedding. The example replaces a single A node by a single B node and transfers all edges from the old node to the new node. Usually, we would implement such a rule by just writing

```
A ==> B;
```

But the example on page 65 does not make use of connection transformations, so we must not use the arrow `==>` for its implementation. The basic rule rather has to be written as

```
A ==>> B;
```

which in isolation would not transfer edges from the old `A` node to the new `B` node. But we may add nested rules to the right-hand side:

```
a:A ==>> b:B
    {
        [
            n:. -e-> a ==>> n -e-> b;
            a -e-> n:. ==>> b -e-> n;
        ]
    };
```

Thus, for each match `a`, we delete `a` and create a new node `b`, but we also execute a rule which transforms each incoming `e`-typed edge of `a` to an incoming edge of `b` (keeping the adjacent node `n`), and a corresponding rule for outgoing edges. These two rules correspond to the productions q_i, q_o of the example on page 65, while the main rule is the common part $q = p_{ij}$. I. e., the amalgamation works by specifying the common part as outer rule so that we may glue together the nested inner rules on the basis of a given binding for the outer part. We could also retype or mirror edges by changing the edge type or direction on the right-hand side. A drawback of this mechanism compared to connection transformations is that if the whole rule is applied in parallel to two adjacent `A` nodes, the result is not what is probably expected. Namely, new edges are created between new `B` nodes and old neighbours `n` of old `A` nodes and not between both new `B` nodes.

The example requires the true parallel derivation mode `PARALLEL_MODE` (without the exclusion of deleted nodes, Sect. 9.1.5 on page 245) for the nested rules as these have to be applied in parallel and each such application deletes `a` on its own.

10.3 Artificial Life

This section contains two examples from artificial life. In fact, although the primary field of application of this work is the description and modelling of botanical real life, artificial life models formed a helpful basis for the abstraction of the requirements for relational growth grammars and, ultimately, for the design of the XL programming language.

10.3.1 Biomorphs

Biomorphs are the creatures created by the well-known "Blind Watchmaker" program, an algorithm written by the zoologist Richard Dawkins [35] and

inspired by the seemingly blind and undirected yet enormously effective phenomenon of evolution by mutation and selection. The algorithm is based on a rather simple genotype-phenotype model and was specified by Dawkins in Pascal [36]. In this original version, it consists of two procedures: the first is responsible for the reproduction of the genotype, thereby producing a number of new individuals and introducing some random chance mutation. The second provides a developmental scheme in which the genotype is expressed following a simplified recursive tree-drawing routine with the genes' values used as parameters (depth of recursion, directions of branches). The genotype consists of nine genes, eight of which have 19 alleles ranging from -9 to $+9$, the ninth a range from 0 to 9, in integral steps. The latter gene determines the depth of recursion. The other genes are translated into components of a matrix which represent the horizontal, respectively vertical offsets in a global coordinate system which are to be used in the subsequent developmental steps of the growing binary tree. The program is started off with an initial set of alleles, which is then modified in the offspring by applying random mutations with given probability.

In our implementation [98, 99], the gene is represented as a sequence of nodes containing **int**-values, i. e., we use the wrapper class `IntNode` of the package `de.grogra.rgg` (Sect. B.10 on page 404). For the whole genome, we declare the class `Genome` which carries its gene nodes as a branch, and we use a node of type `Population` which contains all further nodes of our model. The initialization then looks like

```
Axiom ==> Population Genome [1 1 1 1 1 0 `-1` `-1` 5];
```

The backquotes are required for syntactical reasons, see Sect. 6.5.2 on page 147. Now based on the single genome of the population, but with random mutations of the gene values we create five biomorphs, represented by the type `Biomorph`:

```
[
    p:Population g:Genome ==>>
        p
        for (1:5) (
            Translate(2,0,0) Biomorph [<-encodes- cloneSubgraph(g)]
        );
    {derive();}
    k:int ==>
        if (probability(0.2)) (
            irandom(-9, 9)
        ) else (
            break
        );
]
```

The first rule sets the genome of the biomorphs to identical copies of the original genome, using the method `cloneSubgraph` of `de.grogra.rgg.Library`.

Biomorphs and genomes are linked by model-specific edges of type `encodes`. The second rule then replaces each **int**-wrapper with a probability of 20% by a new such wrapper with a random value between −9 and 9.

Now the recursive tree-drawing routine of Dawkins' model can be easily implemented by instantiation rules (Sect. 6.11.2 on page 182). For this purpose, we define a class

```
module Tip(int[][] d, int depth, int dir)
==> if (depth > 0) (
      Line(0.02 * depth * d[dir][0], 0, 0.02 * depth * d[dir][1], true)
      [instantiate(d, depth-1, (dir+7) % 8)]
      [instantiate(d, depth-1, (dir+1) % 8)]
    );
```

where `d` is Dawkins' offset matrix (to be computed from the genome), `depth` the current recursion depth and `dir` a value indicating the current direction. The instantiation rule corresponds to the one specified by Dawkins, it recursively invokes itself (recall from Sect. 6.11.2 on page 182 that the rule is implemented as a method `instantiate` having the module parameters as its parameters; we could equally well write `Tip(d, ...)` instead of `instantiate(d, ...)`, but this would allocate a new `Tip` instance for each recursion). The drawing is triggered by the instantiation rule of `Biomorph`, initializing `d` with values derived from the genome:

```
module Biomorph extends Sphere(0.2)
==> { Genome g = first((* this <-encodes- Genome *));
      int depth = Math.min(Math.abs(g[8])+1, 6);
      int[][] d = {{-(int)g[1], g[5]}, {-(int)g[0], g[4]},
                   {0, g[3]}, {g[0], g[4]}, {g[1], g[5]}, {g[2], g[6]},
                   {0, g[7]}, {-(int)g[2], g[6]}}; }
    Tip(d, depth, 2);
```

Now the five biomorphs are displayed on the screen, and the user, by applying what Dawkins calls Darwinian (artificial) selection, chooses one of the individuals as parent for the next generation by selecting it with the mouse. To determine which biomorph has been selected, the class `Library` provides the method `isSelected`. We write

```
long c = count((* b:Biomorph, (isSelected(b)) *));
if (c == 1) [
    Population, b:Biomorph <-encodes- g:Genome, (isSelected(b)) ==>>
        ^ Population g;
]
```

I. e., if the user has selected a single biomorph, the whole population is removed, a new one is created, and it contains the genome of the selected biomorph. Now we have the same structure as after initialization, but with a selected genome, and iterate this kind of "artificial evolution" by applying the rules for biomorph creation and genome mutation. By piling-up mutations in a certain direction, which is completely at the bias of the user, a shortcut

is taken through the multidimensional genotypic and phenotypic parameter space with its thousands of millions of combinatorial possibilities, thereby arriving at astonishing biomorphs, i. e., structures that closely resemble organismal morphologies.

As an addition to the original specification by Dawkins, we provide a possibility to select two parent individuals from which offspring is generated using the genetic operation of crossing-over:

```
... // from above
else if (c == 2) [
    Population, b1:Biomorph <-encodes- g1:Genome,
        b2:Biomorph <-encodes- g2:Genome,
        (isSelected(b1) && isSelected(b2)) ==>>
        ^ Population g1 -mate-> g2;
    {derive();}

    int j, k, l, m;
    j k, l m, (* j -axisparent-> -mate-> <-axisparent- l *),
    (* j -align- l *) ==>>
        if (probability(0.3)) (
            j m, l k {println("crossing-over at " + $j.getIndex());}
        )
        else (break);

    g:Genome -mate-> Genome ==>> g;
]
```

The first rule creates a new population which consists of the two selected genomes, connected by an edge of the model-specific type `mate`. Although the left-hand side of this rule is symmetric with respect to `b1`, `b2`, i. e., the match having these and their genomes interchanged is also a match, the right-hand side is executed only for the first found match: already this match adds both biomorphs to the deletion queue, but the default derivation mode excludes nodes from matches which have been added to the deletion queue (Sect. 9.1.5 on page 245). The second rule is the crossing-over rule which was already shown as Ex. 5.3.2 on page 103. `j k` and `l m` stand for two subsequences of genomes. The method `axisparent` of `de.grogra.rgg.Library` follows successor edges in reverse direction until it finds a reverse branch edge whose source node is returned. I. e., in terms of axial trees (Sect. 5.2.1 on page 95), the method returns the parent node of an axis. Now the context `(* j -axisparent-> -mate-> <-axisparent- l *)` ensures that the subsequences `j k` and `l m` stem from the genomes of the two selected parents. The second context `(* j -align- l *)` uses the alignment relation

```
static boolean align(Node a, Node b) {
    return a.getIndex() == b.getIndex();
}
```

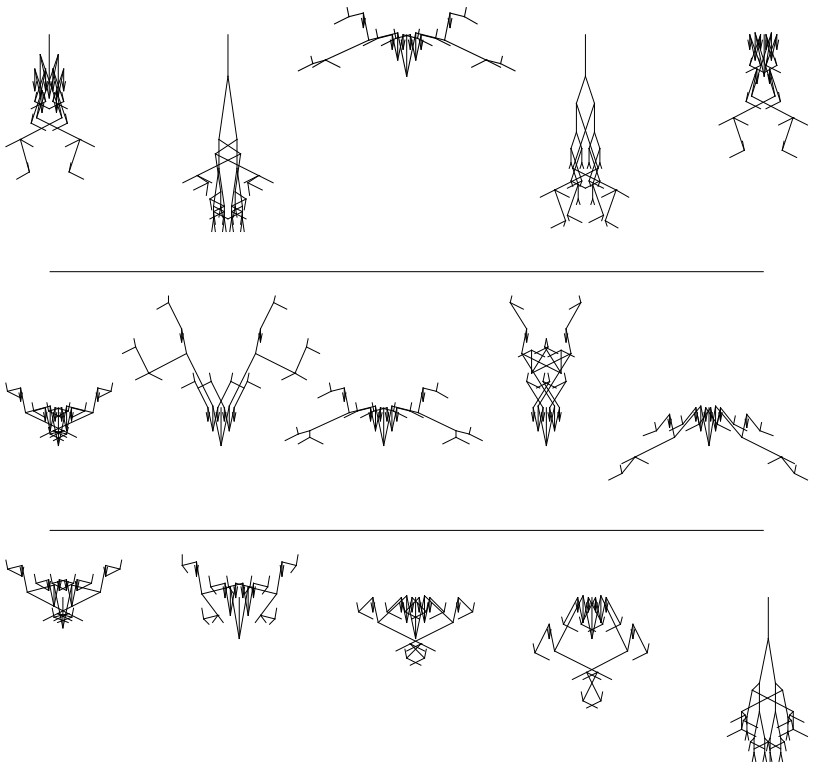


Figure 10.5. Three generations of biomorphs. The biomorph of the first generation drawn by thick lines is the single parent of the next generation. In this generation, two biomorphs were chosen as parents for the third generation

which states that the indices (within the axes) of aligned nodes have to be the same. I. e., if `a` is the p-th gene of the first genome, then also `b` has to be the p-th gene of the other genome. The crossing-over rule is applied with a probability of 30%, otherwise **break** ensures that nothing happens for the current match. The final rule `g:Genome -mate-> Genome ==>> g;` removes the second selected genome, it is applied in parallel together with the crossing-over rule. Figure 10.5 shows three generations of biomorph evolution, the biomorphs drawn with thick lines are the chosen parents for the next generation.

The Biomorph implementation demonstrates the capability of the XL programming language to represent genetic operations like mutation and crossing-over in a natural way. A single crossing-over rule, applied in parallel, models all possible crossing-over exchange events (no crossing-over at all, single, double, triple crossing-over etc.). An implementation of mutation by L-systems is easy (we can mutate the parameter of a gene module), but the implementa-

tion of crossing-over becomes tedious. One solution is to use a single module for both genomes as in

```
G(g0,g1,g2,g3,h0,h1,h2,h3)
```

for a genome length of four, and to specify all possible exchange events explicitly [98, 99]. For example, the rule

```
G(g0,g1,g2,g3,h0,h1,h2,h3) ==> G(g0,g1,g2,h3,h0,h1,h2,g3);
```

models a single crossing-over event between the last two genes, but we need six further rules for the complete set of possible exchanges for a genome length of four. This can be improved by modelling the three possible single exchanges as independent events (with specific probabilities):

```
G(g0,g1,g2,g3,h0,h1,h2,h3) ==>
    {
        boolean x1 = probability(p1);
        boolean x2 = probability(p2);
        boolean x3 = probability(p3);
    }
    G(g0, x1?h1:g1, (x1^x2)?h2:g2, (x1^x2^x3)?h3:g3,
      h0, x1?g1:h1, (x1^x2)?g2:h2, (x1^x2^x3)?g3:h3);
```

However, the problem of a hard-coded genome length remains. This contradicts the rule-based paradigm: its main philosophy is to specify *what* to do with data, but it is left to the formalism (and its implementation) to find the locations *where* to apply the rules. So for a true rule-based crossing-over, we only want to specify a rule for a single crossing-over. Whether there are several possible locations where this rule can be applied in parallel or not has to be detected by the run-time support, not by hand-coding.

10.3.2 Artificial Ants

Artificial ants were already presented in the introduction (Sect. 2.5 on page 15) as an example for agent-based modelling. A simplistic simulation of artificial ants, which nevertheless shows an interesting behaviour, can be implemented easily using the XL programming language [96]. Our 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 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. Cells may also be obstacles.

The rectangular grid consists of cells containing an amount of pheromone which decays step by step. We use the `Cell` class, which was already presented in Sect. 10.1.3 on page 275, and store the pheromone content in the `length`-attribute and the property of being a food source, an obstacle or a normal cell in the `state`-attribute (with values 1, -1, 0, respectively). To speed up the query for neighbourhood in the grid, we use the technique described in Sect. 10.1.3 on page 277, i. e., we create edges of type `neighbour` between neighbouring cells, which are again the eight surrounding cells.

For ants, we use a subclass of `de.grogra.imp3d.objects.Cylinder`, the `length`-attribute being used for the excitation state of an ant. The use of the `length`-attribute in both cases has the advantage of a direct visualization of the corresponding values. Ants have additional attributes `dx`, `dy` to store the current moving direction:

```
public module Ant extends Cylinder(1, 0.1).(setShader(WHITE)) {
    // current moving direction
    int dx;
    int dy;
}
```

At a given point in time, each ant sits on a single cell. This is modelled by a successor edge with the advantage that the 3D visualization of the ant is automatically placed at the correct location, namely at the origin of the local coordinate system of the cell. For the representation of ant memory, the best would be to use memory edges from an ant to all cells which it visited during the last twenty steps. To implement a forgetful memory, these edges have to carry a counter for their lifetime which is initialized with 20 and decremented in each step until it reaches zero, in which case the edge is removed. The graph of GroIMP does not support such attributed edges, but we can use the trick to model such an edge by a triplet of a node with two auxiliary edges. In fact, the implementation of the XL interfaces for the graph of GroIMP already implements this trick (see Sect. B.7 on page 401): defining a node class

```
module Memory(int counter);
```

we may write `a -Memory(20)-> c` on right-hand sides and `a -Memory-> c` on left-hand sides. This implies auxiliary edges of type `EDGENODE_IN_EDGE` from `a` to the memory node and of type `EDGENODE_OUT_EDGE` from the memory node to `c`.

Using the above declarations, the main rule set of the model consists of three rules:

```
m:Memory(c) ==>> if (c > 0) (m {m[counter] :-= 1;});

c:Cell ::> c[length] :-= 0.03 * c[length];

c:Cell a:Ant ==>>
    n:nextCell(c, a) a -Memory(20)-> c
    {
```

```
        a[dx] := (int) Math.round(n[x] - c[x]);
        a[dy] := (int) Math.round(n[y] - c[y]);
        float total = a[length] + 3 * c[state];
        float laidDown = total * 0.1;
        a[length] := total - laidDown;
        c[length] :+= laidDown;
    };
```

The first rule decrements the counter of each memory node if it has not yet reached zero, otherwise the memory node (together with its two auxiliary edges) is removed. The second rule implements an exponential pheromone decay. The last rule specifies ant movement: an ant `a` sitting on cell `c` is moved to the cell `n` which is computed by the method `nextCell` based on `a` and `c`, furthermore a memory edge to the previous cell `c` is created. In the code block, at first the new moving direction of the ant is computed. Then, the total amount of available pheromone of the ant is the sum of the old value and `3 * c[state]`, which is zero for normal cells and 3 for food sources. A fraction `laidDown` is laid down at the cell, the rest is the new pheromone content of the ant (i. e., its excitation state). Note that we use deferred assigments `:=`, `:+=` which take effect later, namely when the derivation is processed (Sect. 9.1.8 on page 251). This ensures that we consistently use the same (old) values within a single step. The central method `nextCell` looks as follows:

```
static Cell nextCell(Cell c, Ant a) {
    float dx, dy;
    Cell next = selectWhereMax
        ((* c -neighbour-> n:Cell,
            ((n[state] >= 0) && empty((* a -Memory-> n *))) *),
         (dx = (n[x] - c[x]) - a[dx], dy = (n[y] - c[y]) - a[dy],
          rank(n[length], dx*dx + dy*dy)));
    return (next != null) ? next : c;
}
```

The next cell is chosen among all neighbours `n` of the current cell `c` which are no obstacles (`n[state] >= 0`) and which are not recorded in the memory of the ant. The aggregate method `selectWhereMax` (Sect. 6.4.3 on page 140) chooses the candidate `n` for which the second expression is maximal. Here this expression is the result of the method `rank` and based on the amount of pheromone at `n` and the square of the distance between the previous moving direction and the new one that would result from moving to `n`. The implementation of `rank` ranks those candidates higher which have a higher amount of pheromone or whose direction is closer to the previous direction, but it also contains a random effect:

```
static float rank(float pheromone, float deltaDirSquared) {
    float t = random(0, 1);
    return (pheromone + 0.14 * (8 - deltaDirSquared)) + 0.7 * t;
}
```

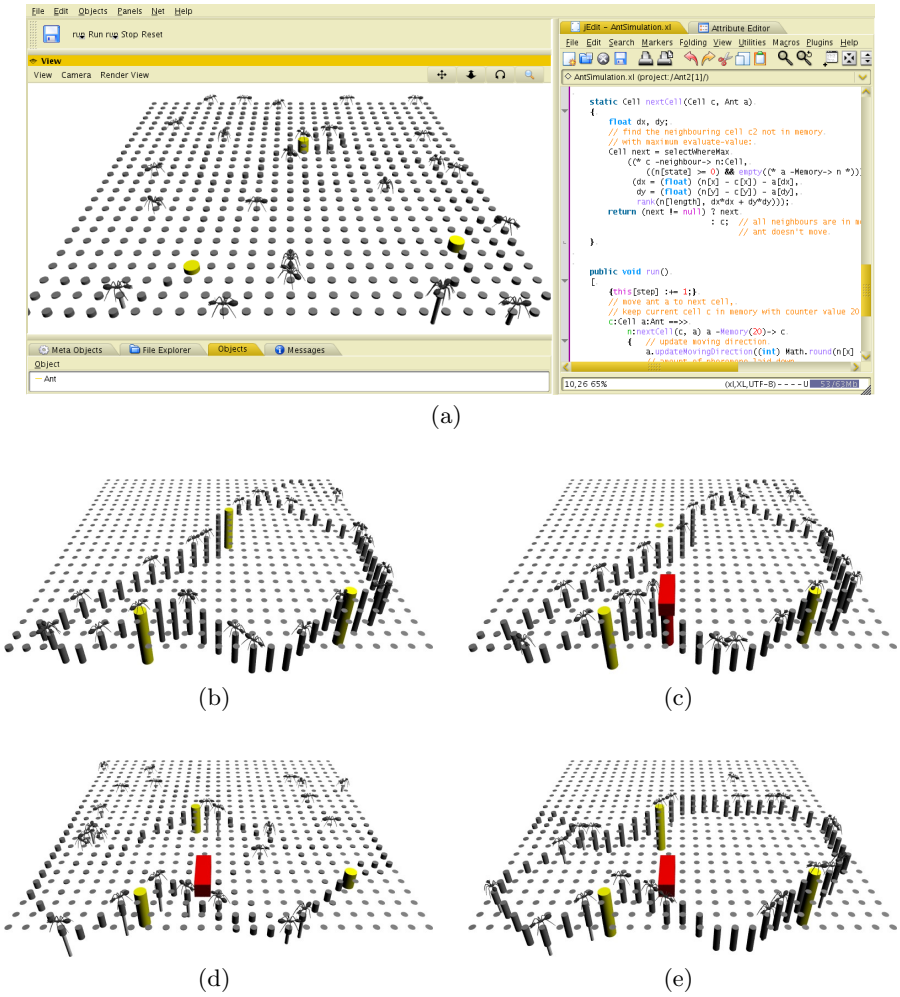


Figure 10.6. Execution of Ant Model: (**a**) early situation with ants exploring the world, note the Objects panel with an entry named `Ant` for the predefined ant geometry; (**b**) stable path through all food sources (yellow cylinders) after 680 steps; (**c**) addition of a new food source and an obstacle (red boxes), deletion of a food source; (**d**) some ants have found the new food source after 200 steps; (**e**) stable path through all food sources after another 200 steps

Its parameters have been chosen manually so that the behaviour of ants is at the border between erratic and too deterministic. As a result, ants of our model are able to find short paths through all food sources after several steps, and even if such a stable path has been found, the addition or deletion of food sources is detected by the ants after a while so that they find a new path through the food sources. Figure 10.6 shows this behaviour. For a nice visualization, we add an ant-shaped geometry by means of an instantiation rule in the class `Ant`:

```
module Ant extends Cylinder(1, 0.1).(setShader(WHITE)) {
    ... // declarations from above

    const Reference geometry = reference("Ant");
    const Null[][] transforms = new Null[3][3];
    static {
        ... // initialization of transforms
    }
} ==> `transforms[dx+1][dy+1]` P(DARK_GRAY) geometry;
```

I. e., an ant is not only drawn as a cylinder, it also instantiates a sequence of a coordinate transformation `transforms[dx+1][dy+1]`, a `P`-command to set the shader to `DARK_GRAY`, and finally the node `geometry` which is a reference to the object named `"Ant"`. Such an object is defined in the Objects panel of GroIMP (see Fig. 10.6(a) on the preceding page and Sect. A.5 on page 387), here we import a manually created mesh of polygons (Sect. A.6 on page 388, Sect. B.13.5 on page 411). The array `transforms` has to be initialized such that, when indexed by the pair $(\mathtt{dx}+1, \mathtt{dy}+1)$, the transformation places the referenced geometry at the right location and rotates the ant such that it looks in moving direction. The backquotes around the transformation serve as parentheses, they are necessary for syntactical reasons as the brackets would otherwise be interpreted as branch delimiters (Sect. 6.5.2 on page 147). As a further step, we turn the whole grid upside down such that the cylinders of cells and ants point downwards, while the ant geometry is placed above. Finally, by a second instantiation rule we represent obstacle cells by boxes:

```
module GridCell(super.state) extends Cell(0, 0, 0, state)
    ==> if(state < 0) (RU(180) Box);
```

The ant model benefits from the possibility to establish arbitrary edges between nodes. We use edges to mark the current cell of an ant, thereby automatically defining the 3D position of the ant, to encode the neighbourhood in a time-efficient way, and to represent the memory of an ant consisting of the last twenty cells visited. All ingredients of the model can be coded in a natural way, resulting in a very short and concise specification. Apart from the initialization and the handling of ant geometry, the model consists of less than 40 lines. This is not possible with L-systems, the major reason being their pure string data structure.

10.4 Artificial Chemistry

A calculus which got some attention in the artificial life community is artificial chemistry (see [61]), i. e., the attempt to model the dynamics of large numbers of artificial "molecules" (which can be numbers, code fragments, graphs or other abstract objects with nontrivial pairwise interaction) in a virtual solution. We present two simple examples: a rather abstract prime number generator, and a more physical model of atoms which roam around and, when they are in proximity, create chemical bonds, leading to polymer-like molecules.

10.4.1 Prime Number Generator

A "chemical" prime number generator [176] consists of a soup of positive integers as molecules and a single interaction in the case of a collision: if the integer values of two colliding molecules are given by a, b such that a divides b, then a remains in the soup, but b reacts to $\frac{b}{a}$. Using XL, we initialize such a soup as a set of wrapper nodes for random **int**-values:

```
Axiom ==>> for(1:20) (^ irandom(2, 200),);
```

This appends the wrapper nodes at the root (symbol `^`), the comma is necessary to separate the structure created by one execution of the body from the next one (Sect. 6.5.4 on page 151). The reaction is given by

```
(* a:int *), b:int, ((a != b) && ((b % a) == 0)) ==> `b / a`;
```

`a` is in the context of the rule so that it remains in the soup. Only `b` is replaced by the quotient if the application condition holds. The application of this rule benefits from the default derivation mode (Sect. 9.1.5 on page 245) which excludes nodes which are already enqueued for deletion from subsequent matches. For example, if the soup contains the numbers $2, 3, 6$, then both $(2, 6)$ and $(3, 6)$ are matches for $(\mathtt{a}, \mathtt{b})$, but only the first found match is actually used for rule execution as it enqueues 6 for deletion. Without this mechanism, the parallel application would create two new molecules out of a single one.

The following shows a random initial content of the soup and its development in five steps to a final state (i. e., a fixed-point). Prime numbers are marked with an asterisk.

```
57 74 169 115 20 66 87 10 133 48 23* 165 36 112 99 103* 176 74 84 129
57 74 169 5* 2* 66 87 10 133 48 23* 165 36 112 99 103* 176 74 84 129
57 37* 169 5* 2* 33 87 5* 133 24 23* 33 18 56 99 103* 88 37* 42 129
57 37* 169 5* 2* 33 87 5* 133 12 23* 33 9 28 3* 103* 44 37* 21 129
19* 37* 169 5* 2* 11* 29* 5* 133 4 23* 11* 3* 14 3* 103* 22 37* 7* 43*
19* 37* 169 5* 2* 11* 29* 5* 19* 2* 23* 11* 3* 2* 3* 103* 2* 37* 7* 43*
```

As we can see, all but one non-prime numbers have reacted to prime numbers. The non-prime number $169 = 13^2$ is still in the soup as its single divisor 13 is not present.

Even this simple example would not be possible with L-systems without global sensitivity in the presented, concise way. While for the former examples the string data structure of L-systems turned out to be too limited, it contains too much information for this example: we do not need any neighbourhood information, just the presence of molecules in a set. The rule simply queries for a pair of nodes, ignoring their topological relation. I. e., from a topological point of view, both nodes may have a "global distance" so that (locally) context-sensitive L-systems are not sufficient.

10.4.2 Polymerization Model

Our second model of artificial chemistry consists of spherical atoms (or monomers) enclosed in a two-dimensional rectangular region [109]. They move in random initial directions with constant velocity, the boundaries of the region are reflecting. Atoms may have chemical bonds to other atoms; such bonds are established when two atoms are in proximity. This leads to a polymerization of the initially unbonded atoms. With respect to the (mechanical) kinetics, a bond behaves like a spring, its rest position being given by the situation when the bond was created.

For the atoms, we define a subclass of `Sphere` with additional attributes for the current velocity and the mass:

```
module Atom(float vx, float vy, float mass) extends Sphere
    .(setShader(LIGHT_GRAY))
{
    Atom(float x, float y, float vx, float vy, float mass) {
        this(vx, vy, mass);
        setRadius((0.001 * mass)**(1.0/3));
        setTranslation(x, y, 0);
    }
}
```

The rule for movement with constant velocity (Newton's first law) is simply

```
a:Atom ::> {a[x] :+= DELTA_T * a[vx]; a[y] :+= DELTA_T * a[vy];}
```

where `DELTA_T` is a constant defining the time step of the Euler-like numerical integration. We also have a rule implementing the reflection at the boundaries of the rectangular region $[-1, 1] \times [-1, 1]$:

```
a:Atom ::> {
    if (((a[x] >= 1) && (a[vx] > 0))
        || ((a[x] <= -1) && (a[vx] < 0))) {
        a[vx] = -a[vx];
    }
    if (((a[y] >= 1) && (a[vy] > 0))
        || ((a[y] <= -1) && (a[vy] < 0))) {
        a[vy] = -a[vy];
    }
}
```

For the representation of bonds, we define a second node class which carries attributes related to spring-like behaviour:

```
module Bond(float springRate, float friction, float dx, float dy);
```

As a bond relates two atoms, it can be seen as an attributed edge in terms of the model, although it is a node in the underlying graph of GroIMP. Just like for the memory relation in the ant example (Sect. 10.3.2 on page 288), this detail is hidden in the syntax, i. e., we can write `-Bond->`.

The single "chemical" reaction takes place when two atoms are closer than the sum of their radii, multiplied by 1.5:

```
a1:Atom, a2:Atom,
((a1 < a2) && (distance(a1, a2) < 1.5 * (a1[radius] + a2[radius]))
 && empty((* a1 -Bond-> a2 *))) ==>>
    a1 -Bond(1, 0.1, a2[x] - a1[x], a2[y] - a1[y])-> a2;
```

To compute the distance, we use the corresponding method in the class `Library`. We further add the condition `a1 < a2` to prevent the match with both atoms exchanged (note that `Library` defines a total order on nodes by overloading the comparison operators), and the condition that there does not yet exist a bond from `a1` to `a2`. On the right-hand side, we create a bond whose parameters define its properties as a spring.

For bonds, we have to add another rule which computes the resulting spring force and computes changes in velocity due to Newton's second law. Given a spring constant k, a friction r and a neutral extent $\mathbf{d}$ of the axis of the spring, the force for a current extent $\mathbf{x}$ and a relative velocity $\mathbf{v}$ of the end point is given by

$$\mathbf{F} = D(\mathbf{x} - \mathbf{d}) + r\mathbf{v} \; .$$

Dividing the force by the individual masses, we obtain the accelerations and, using the discrete time step `DELTA_T`, the increments in velocities:

```
l:Atom -Bond(rate, fr, dx, dy)-> r:Atom ::> {
    float fx = rate * (r[x] - l[x] - dx) + fr * (r[vx] - l[vx]);
    float fy = rate * (r[y] - l[y] - dy) + fr * (r[vy] - l[vy]);
    l[vx] :+= DELTA_T * fx / l.mass;
    l[vy] :+= DELTA_T * fy / l.mass;
    r[vx] :+= DELTA_T * -fx / r.mass;
    r[vy] :+= DELTA_T * -fy / r.mass;
}
```

To visualize the bonds, we use the technique of the Sierpinski example to visualize edges by thin cylinders (Sect. 10.1.2 on page 274). Here, we add an instantiation rule to the definition of `Atom`:

```
module Atom ... // see above
==> for ((* this -Bond-> a:Atom *)) (
        [
            Cylinder(0, 0.03).(setEndPoints(ORIGIN, a - this),
                               setShader(YELLOW))
```

```
        ]
    );
```

Figure 10.7 shows an initial state with ten atoms, two intermediate steps and the final polymer in which all atoms are bonded. Like the previous example of the prime number generator, this example starts with an unstructured set of atoms and considers pairwise interactions, now based on the geometric quantity of distance. But when a reaction happens, the example adds structure to the set by means of bond edges. This again shows the versatility of our approach, here in combination with a library of geometric functions. We can also see the benefits of deferred assignments: the physical laws of kinetics can be written down (in a time-discretized form) in a very natural way. There may be several simultaneous contributions to the same variable, but these do not interfere with each other as it is only after the execution of all rules that the run-time system merges the simultaneous individual changes to a resulting change (Sect. 9.1.8 on page 251).

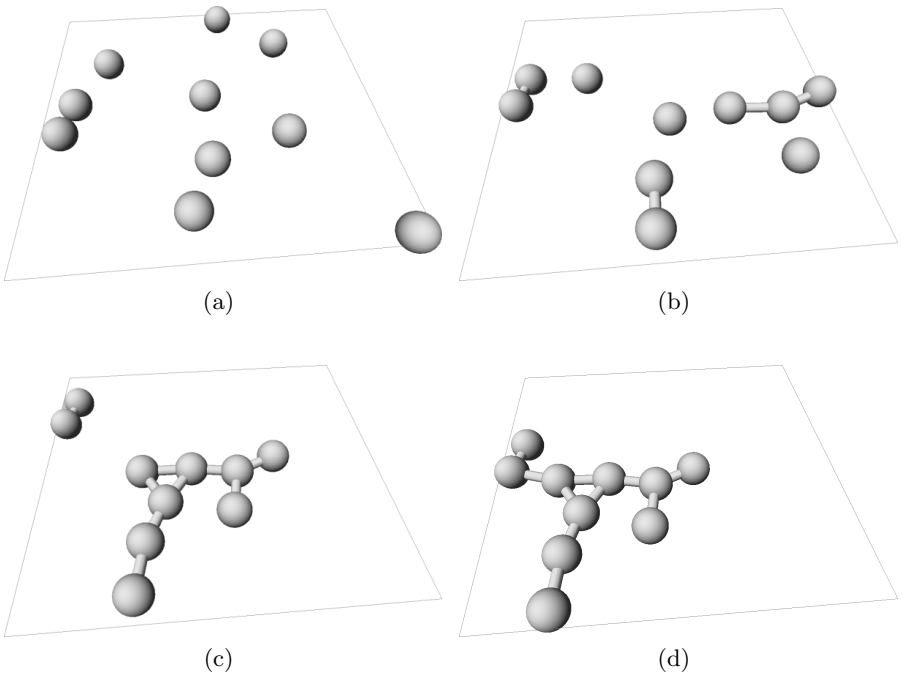


Figure 10.7. Polymerization: (**a**) initial situation with unbonded atoms; (**b**) some bonds have been created; (**c**) two polymers remaining; (**d**) final single polymer

10.5 Virtual Plants

The previous examples were not taken from the realm of plants, but from several other fields of application. The main intent was to highlight the diverse spectrum of possibilities of relational growth grammars and the XL programming language, embedded in GroIMP, at simple yet instructive examples. But the examples did not make use of the power of L-systems. Returning to botany with a collection of virtual plants or even virtual forest stands, this aspect of relational growth grammars becomes essential, but now, within the extended framework, the expressiveness is considerably enhanced. As the models are relatively complex, we do not give the complete source code, but explain the main ideas and rules. However, the complete source code of most of the models is available as part of the example gallery of GroIMP distributions. In addition, all presented models of this section were published in some form. The author expresses his gratitude to the co-authors of the publications for the fruitful cooperation, and, where the author was not involved in the development of the models, to the authors for kindly providing their models.

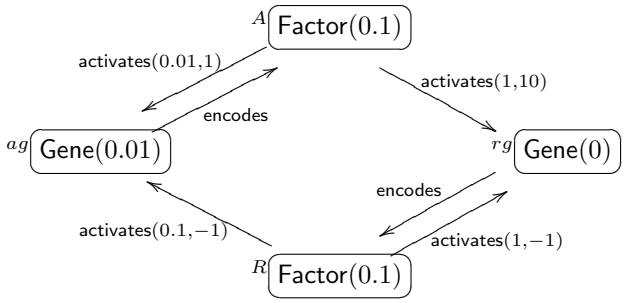
10.5.1 ABC Model of Flower Morphogenesis

As an example of the new possibilities that arise from the combination of L-system-like growth rules with more general graph rewriting rules of relational growth grammars, let us consider flower morphogenesis controlled by a genetic regulatory network. The ABC model of flower morphogenesis describes this process using three genes A, B, C with corresponding transcription factors [27]. According to the dynamics of a regulatory network, these factor concentrations change in time, thereby leading to a time-variant expression of functions that control morphogenesis. Depending on the current state, different flower organs are formed. Finally, the sequence and type of organs formed constitutes the mature flower.

From a structural point of view, flower morphogenesis can be described by L-system rules, but for the ABC mechanism these rules have to be influenced by the state of a regulatory network, i. e., by a graph. In Sect. 3.15.4 on page 38, we briefly presented the program L-transsys which addresses the requirements of such an integrated model of flower morphogenesis [94]. However, both model levels, the L-system controlling morphogenesis and the regulatory network, reside in their own, separate domains of the underlying formal language. Using relational growth grammars and the XL programming language, we can concisely specify both levels in the same language.

Our implementation of the ABC model [96] is a direct translation of the implementation of [94]. At first, we have to encode the gene regulatory network. Such a network consists of nodes for genes and transcription factors. Genes encode factors, so there may exist edges from genes to factors, and factors activate or repress genes. The latter is best represented by edges from factors to genes, which carry the two attributes K_m, v_{max} of Michaelis-Menten

kinetics. Genes have a single attribute c which specifies a rate of production for their encoded factors from some external source. Factors have a single attribute d for their decay. The simple oscillatory network from Sect. 3.15.4 on page 38 can be depicted by



where we have modelled repressions by activation edges with negated value of v_{max}. Declaring classes for genes, factors and activation edges and a simple edge type for 'encodes' edges

```
module Gene(double constitutive);
module Factor(double concentration, double decay);
module Activates(double spec, double max);
const int encodes = EDGE_0;
```

we set up the example network by

```
... ==>> ...
    ag:Gene(0.01) -encodes-> A:Factor(0, 0.1),
    rg:Gene(0) -encodes-> R:Factor(0, 0.1),
    A -Activates(0.01, 1)-> ag, A -Activates(1, 10)-> rg,
    R -Activates(0.1, -1)-> ag, R -Activates(1, -1)-> rg;
```

In the same way, but with six genes (A, B, C and three auxiliary genes) and six factors we specify the original ABC network of [94]. The dynamics of such a network as given in [94] can be specified by two execution rules. The first one is responsible for the decay:

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

In principle, this should be a numerical solver for the ordinary differential equation of decay; however, the original model, which we want to reimplement here, already uses this discrete form. The second rule is responsible for the Michaelis-Menten kinetics of the regulatory network [11]. Given a factor f which is encoded by a gene g, we have to consider all factors a activating or repressing g. The individual contribution to the change in the concentration of f is

$$\Delta_{a\to g\to f} = \frac{v_{max} c_a}{K_m + c_a}$$

where c_a is the concentration of a and v_{max}, K_m the Michaelis-Menten constants of activation or repression. The model of [94] adds all contributions

and the constitutive effect of g, and uses the resulting change, where negative values are cut off:

```
f:Factor <-encodes- g:Gene(gc) ::> f[concentration] :+= Math.max(0,
    sum(((* Factor(ca,) -Activates(s,m)-> g *), m*ca / (s+ca))) + gc);
```

When using the ABC network with the original values, we obtain the development of concentrations of the factors for A, B, C as shown in Fig. 10.8. Now according to the ABC model of flower morphogenesis, these concentrations determine the type of flower organ to which the flower meristem differentiates. In [94], the following thresholds were used: $b > 80, c > a$ leads to a stamen, $b > 80, c \leq a$ to a petal. When $b \leq 80, a > 80, c > 80$, the meristem differentiates to a shoot, for $b \leq 80, a > 80, c \leq 80$ to a sepal. Finally, $b \leq 80, a \leq 80, c > 80$ yields a carpel and $b \leq 80, a \leq 80, c \leq 80$ a pedicel. In Fig. 10.8, the organ type for the corresponding set of concentrations is indicated. The resulting order (pedicel, sepal, petal, stamen, carpel) is the usual order of the wild type.

To implement this morphogenesis, we use a simple L-system for a flower meristem, but this meristem is connected to the gene regulatory network and creates organs depending on the current concentrations. The meristem itself is defined as

```
const int SHOOT = 0, PEDICEL = 1, SEPAL = 2, PETAL = 3,
    STAMEN = 4, CARPEL = 5, TERMINATE = 6;

module Meristem(int type, float mass);
```

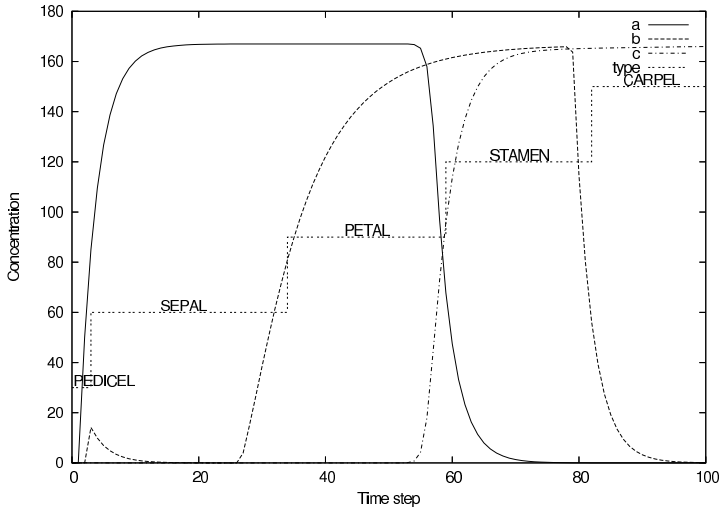


Figure 10.8. Development of concentrations and resulting type of flower organ

where its attribute `type` is one of the constants for organ types. On initialization of the model, we create a single meristem which bears the factors a, b, c, cc of the network (cc is one of the three auxiliary factors of [94]):

```
Axiom ==>>
    ... , // creation of the regulatory network, see above
    ^ Meristem(SHOOT, 1) [-factors-> a b c cc];
```

This meristem then creates new organs which are differentiated according to the current concentrations:

```
m:Meristem(type, mass)
    (* -factors-> Factor(a,) Factor(b,) Factor(c,) Factor(cc,) *) ==>
    {   // Classification of next organ type
        int t = ((c > 80) && (cc > 1)) ? TERMINATE
            : (b > 80) ? ((c > a) ? STAMEN : PETAL)
            : (a > 80) ? ((c > 80) ? SHOOT : SEPAL)
            : (c > 80) ? CARPEL : PEDICEL;
    }
    if (t == type) (
        {m[mass] :+= 1;}
        break
    ) else (
        switch (type) (
            case SHOOT:
                Cylinder(0.45*mass, 0.6).(setColor(0x808000))
                break
            case PEDICEL:
                Cylinder(0.18*mass, 0.6).(setColor(0xb8e070))
                break
            case SEPAL:
                P(0xb8e070) M(0.1)
                for (0:3) (RH(90) [RL(80) M(0.5) Scale(3) sepal])
                break
            case PETAL:
                P(0xd7d9cb) RH(45)
                for (0:3) (RH(90) [M(-0.5) RL(30) M(0.9)
                                   RH(180) Scale(5) petal])
                break
            case STAMEN:
                P(0x909778) M(0.1)
                for (0:5) (RH(60) [RL(20) M(0.9) Scale(2) stamen])
                break
            case CARPEL:
                P(0xa8b204) M(0.1)
                for (0:1) (RH(180) [M(0.5) RL(-5) Scale(3) carpel])
                break
        )
        if (t != TERMINATE) (
            {m[type] := t; m[mass] := 1;}
```

```
            m
        ) else (
            cut
        )
    );
```

To be more precise, if the new type `t` is the same as the previous one `type` (which is stored as part of the state of the meristem), the organ is not yet created, but its prospective mass is incremented. If the new type differs from the previous one, an organ according to the previous type is created, and the new type is recorded in the meristem. If the special condition for termination holds, which makes use of the auxiliary factor *cc*, growth terminates by cutting away the meristem together with its network (note that the invocation of `cut` corresponds to the cut-operator % of L-systems, see Sect. 9.1.4 on page 244).

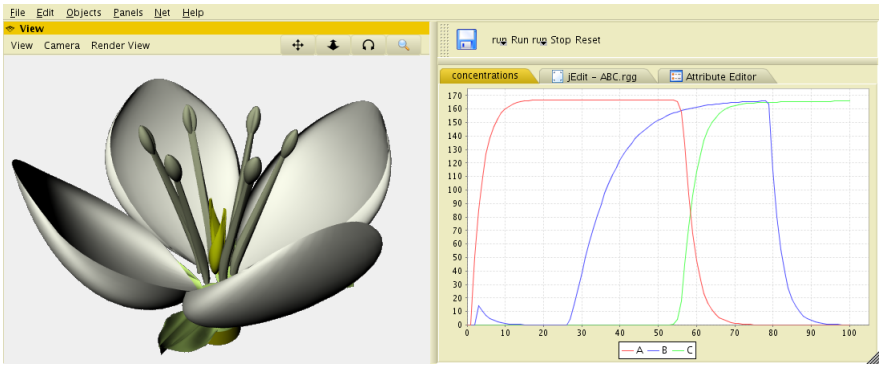
The individual organs are represented by nodes with suitable 3D geometry. For sepals, petals, stamens and carpels, we use Bézier patches which were modelled interactively. Like the ant geometry in Sect. 10.3.2 on page 287, these can be imported in the Objects panel and then used via a `Reference` node (Sect. B.13.5 on page 411):

```
const Reference sepal = reference("sepal");
...
```

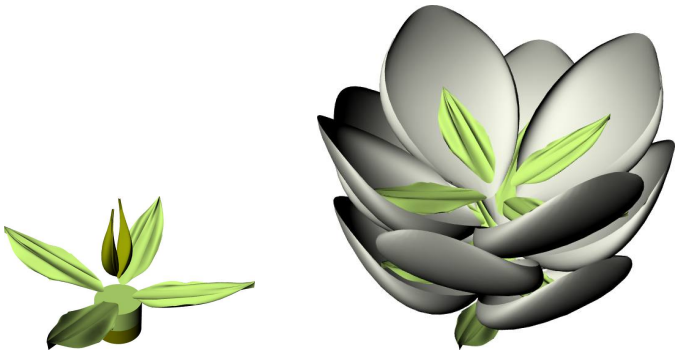
The final outcome of the model is shown in Fig. 10.9(a) on the next page. There we also see a plot of the development of concentrations. This can be obtained within GroIMP by the built-in chart functionality based on the JFreeChart library [137]:

```
const DatasetRef concentrations = new DatasetRef("concentrations");

protected void init() {
    concentrations.clear()
        .setColumnKey(0, "A").setColumnKey(1, "B").setColumnKey(2, "C");
    chart(concentrations, XY_PLOT);
    ... // initialization of ABC model
}

public void run() {
    ...
    m:Meristem(type, mass) // growth rule from above
    (* -factors-> Factor(a,) Factor(b,) Factor(c,) Factor(cc,) *) ==>
        {   concentrations << a << b << c;
            ... // see above
        }
        ... ;
}
```

On initialization, the dataset referred by `concentrations` is cleared, and the labels for its three columns are set. Then in each step of the model, a new row



(a)



(b) (c)

Figure 10.9. Simulation of ABC model: (**a**) GroIMP window showing wild type and plot of development of concentrations; (**b**) "loss-of-B" mutant with petals and stamens missing; (**c**) "gain-of-A" mutant with sepals and petals repeating

is added which contains the current concentration values `a`, `b`, `c`. The addition of the values makes use of the overloading of the operator `<<`.

It is easily possible to modify the gene regulatory network, either by modifying its parameters or by even modifying its topology. Such mutations also occur in nature and then lead to a morphogenesis different from the wild type. Figure 10.9(b) shows a "loss-of-B" mutant where the `encodes`-edge from gene B to its factor has been removed. Figure 10.9(c) shows a "gain-of-A" mutant where the `constitutive`-attribute of gene A has been set to 500. Both mutants are qualitatively correct [94], i. e., they can also be found in nature.

The ABC example shows how we can easily represent processes at different scales and of different nature within the framework of relational growth grammars. Both network dynamics and morphogenesis fit well into this framework, and their combination gives rise to an interesting functional-structural model

of a flower. The advantage over the original approach of [94] is that the latter used the specialized system L-transsys with two separated language domains, whereas we concisely specify the complete model in a single framework.

A disadvantage of this approach to model gene regulatory networks is that we mix the specification of the network with instances where the network shall be active. E. g., factors have both the attribute `decay`, which is a common property of the network and valid for all locations where this network occurs, and the attribute `concentration`, which represents the current state at a single location. If the network shall be used at more than one place in our model (e. g., a single plant with several flowers or even several plants), we would have to duplicate not only the state variables, but also the common specification of the network. A cleaner solution would be to remove the attribute `concentration` from factors, and to regard the network as a specification of a kind of rule that can be applied to entities like meristems or cells and modifies concentration values within these entities. We will see such a solution in the next example.

10.5.2 Barley Breeder

The next example is a multiscaled ecophysiological model of barley development [19, 20]. It was developed as part of the work within the research group "Virtual Crops" consisting of a number of scientists of different plant and information science disciplines. Similar to the ABC model, the barley model consists of a set of morphogenetic rules having the overall shape of L-system rules, and of a regulatory network at a lower level. Here, the network is a metabolic regulatory network which simulates the biosynthesis of gibberellic acid (GA_1). This plant hormone is responsible for internode elongation both in nature and in our model.

A further ingredient of our model is the representation of a simplified barley genome. Like for the biomorph example (Sect. 10.3.1 on page 282), a set of barley individuals is presented to the user, and the user can choose one or two as parents for a next generation, which again undergoes the genetic operations of mutation and crossing-over. Thus, the model could be used as a basis for a "breeder's tool" where a plant breeder can manually or automatically find a breeding path from a given input population to a desired phenotype.

The implementation of the genetic operators and the user selection is very similar to the biomorph example. The main difference is that the genome of barley is diploid, i. e., we have two chromosome sets instead of a single one. Therefore, we add a further node class `ChromoSet` representing single chromosome sets. Furthermore, the class `Genome` of the biomorph example is replaced by a class `GenEnv` which also contains some environmental parameters, and instead of `Biomorph`, we use `Individual` as root node for barley plants. From an individual we reach the genome-and-environment node by an edge of type `env`:

```
module ChromoSet;
module Individual extends TextLabel;
module GenEnv(int time, int maxRank, double[] tsum);

const int env = EDGE_0;
```

The initialization of a single genome then looks like

```
Axiom ==> Population GenEnv [ChromoSet [0 1 1 0 0 1 1]
                             ChromoSet [0 1 1 0 1 0 0]];
```

The seven genes (now with the only possible values 0, 1) stand for *cer-ze* (responsible for a wax layer on the epidermis, conferring a blue-green colour), *lks2* (awn length), *vrs1* (number of spikelet rows), *Zeo* (dwarfing gene affecting lengths of all internodes), *Blp* (lemma and pericarp colour), *glo-b* (globe-shaped grain) and *cul2* (uniculm, i. e., branching is suppressed).

The genome is reproduced in the same way as for biomorphs, leading to five barley individuals, and their initially identical genomes are mutated randomly. What has to be handled differently is the case of crossing-over for sexual reproduction. This happens before the actual reproduction as part of the meiosis when diploid cells divide into haploid cells (i. e., with only a single chromosome set). The reproduction then combines two haploid cells (one of each parent) into a single diploid cell. In our implementation, we create a `co` edge between the two chromosome sets of each selected parent and already combine the first chromosome sets to the new diploid genome of the offspring:

```
Population, i1:Individual [-env-> GenEnv [c11:ChromoSet c12:ChromoSet]],
            i2:Individual [-env-> GenEnv [c21:ChromoSet c22:ChromoSet]],
            ((i1 < i2) && isSelected (h1) && isSelected (h2)) ==>>
    ^ Population GenEnv [c11 [-co-> c12] c21 [-co-> c22]];
```

We then apply the crossing-over rule from the biomorph example, but now with a position-dependent probability of crossing-over to account for different gene distances and with `co` instead of `mate` [99, 19]. Finally, we remove the second chromosome set:

```
c:ChromoSet -co-> ChromoSet ==>> c;
```

The usage of the actual gene values has to combine the corresponding values of both chromosome sets. In case of dominant inheritance, the presence of a 1-value in at least one set suffices to activate the corresponding function. If `g` is the `GenEnv`-node, this is specified by expressions like

```
if (((int)g[0][6] | (int)g[1][6]) != 0) {
    ... // function of uniculm gene (index 6)
}
```

The genetic level is the basic level of our model. On top of it, we specify the metabolic level. For this level, we choose a simplified network of GA_1 (gibberellic acid) synthesis. The network is shown in Fig. 10.10 on the following page. GA_{19} is produced in apical meristems, the amount being a function of

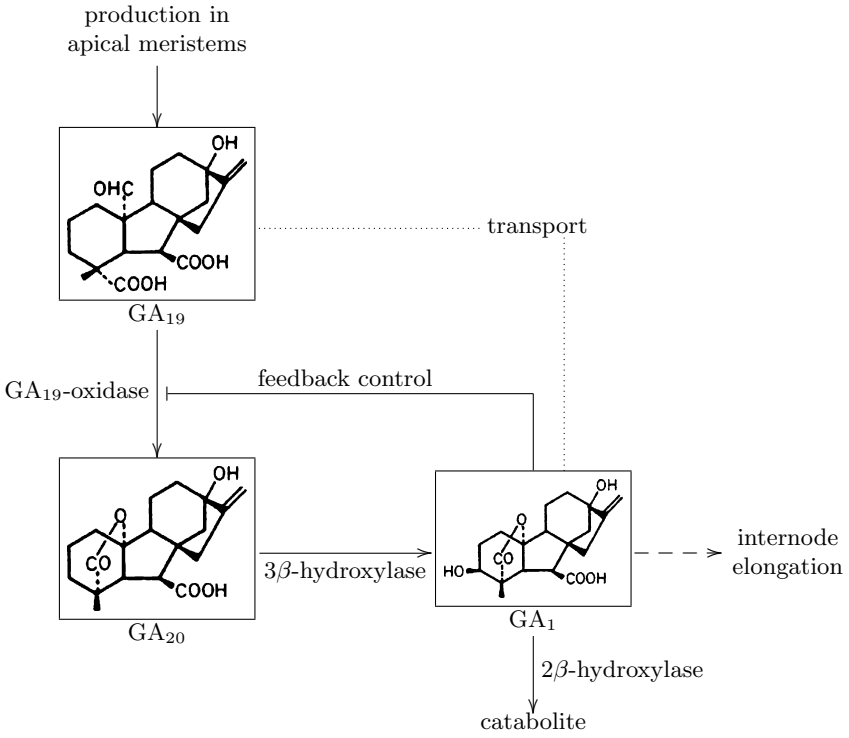


Figure 10.10. Simplified scheme of GA biosynthesis pathway as used in the model

time and of some control by the dwarfing gene *Zeo*. Catalyzed by the enzymes GA_{19}-oxidase and 3β-hydroxylase, GA_{19} is converted via the intermediate GA_{20} into the bioactive GA_1 which controls internode elongation and is itself degraded by 2β-hydroxylase into an inactive catabolite. GA_1, on the other hand, competes with GA_{19} for the binding site on GA_{19}-oxidase, thus competitively inhibiting the production of GA_{20} and, ultimately, its own production. As a first simplified hypothesis of metabolite transport, we also include basipetal transport of GA_{19} and acropetal transport of GA_1 along the stem.

For the implementation of the metabolic network we could use the approach of the previous ABC model of flower morphogenesis. However, as we pointed out at the end of the description of the ABC model, this is disadvantageous if there are several instances of the network as we would have to duplicate the whole network specification for each instance. Therefore, we use another technique. We define an abstract class for substances and concrete subclasses for each metabolite GA_1, GA_{19} and GA_{20}:

```
module Substance (double concentration);
module GA1(super.concentration) extends Substance;
```

```
module GA19(super.concentration) extends Substance;
module GA20(super.concentration) extends Substance;
```

Now each internode and each meristem is a place where reactions according to the metabolic network take place and which participates in the transport of substances. For internodes and meristems, we use a common base class `Organ` whose instances bear one node for each metabolite. The specification of the Michaelis-Menten kinetics [11] from GA_{20} to GA_1 is then given by

```
Organ [s:GA20] [p:GA1] ::> michaelisMenten(s, p, 0.2 * DELTA_T, 1);
```

I. e., for each organ which contains the substances GA_{20}, GA_1, we invoke a method for Michaelis-Menten reaction. This method is specified as part of the model, but could also be provided as part of a common library due to its generic nature:

```
void michaelisMenten(Substance s, Substance p, double max, double km) {
    double r = max * s[concentration] / (km + s[concentration]);
    s[concentration] :-= r;
    p[concentration] :+= r;
}
```

`DELTA_T` is the time step for discretization. The choice of the parameters $v_{max} = 0.2, K_m = 1$ (and of all following parameters) was done by "manual optimization", i. e., such that the outcome looks useful. This means that, whatever we obtain as a result from our model, can only be seen as a proof of concept for the ability of relational growth grammars to represent a functional-structural plant model from the genetic level via the metabolic level up to the morphologic level. We do not claim that this is a physiologically validated model.

The reaction from GA_{19} to GA_{20} with competitive inhibition by GA_1 is modelled very similar, now invoking another method which implements the numerics of competitive inhibition [11]:

```
Organ [s:GA19] [p:GA20] [f:GA1] ::>
    competitiveInhibition(s, p, f, 0.1 * DELTA_T, 2, 0.4);
```

Both the production of GA_{19} by apical meristems and the catabolism by 2β-hydroxylase are assumed to depend on time and, for production of GA_{19}, also on the genotype. Time (measured in days) and genotype can be obtained by the `GenEnv`-node of the barley in question. We write

```
Meristem [ga:GA19] -ancestor-> Individual -env-> g:GenEnv ::>
    ga[concentration] :+= ga19Prod(g[time], g) * DELTA_T;

Organ [s:GA1] -ancestor-> Individual -env-> g:GenEnv ::>
    catabolism(s, ga2betaActivity(g[time]) * DELTA_T, 1);
```

with the functions for GA_{19} production and 2β-hydroxylase activity

```
static double ga19Prod(GenEnv g) {
```

```
    return ((g[time] > 20) && (g[time] < 40))
        ? (((int)g[0][3] | (int)g[1][3]) == 0) ? 0.5 : 0.1
        : 0;
}

static double ga2betaActivity(double t) {
    return (t < 60) ? 0.8 : 0;
}
```

`ancestor` refers to a method in `de.grogra.rgg.Library` which finds the closest ancestor of a given node that has a specified type. Here, we are looking for the `Individual` as the ancestor of all of its organs.

From Fig. 10.10 it remains to implement the transport. In our model, GA_1 is transported acropetally, i. e., from the root to the tips. Furthermore, we assume that this transport is only active during the first forty days:

```
Organ [a:GA1] -ancestor-> Organ [b:GA1]
        -ancestor-> Individual -env-> g:GenEnv, (g[time] <= 40) ::> {
    double r = 0.04 * b[concentration] * DELTA_T;
    b[concentration] :-= r;
    a[concentration] :+= r;
}
```

This rule specifies that for each organ with an ancestor organ, GA_1 is moved from the ancestor to the organ, the rate being the concentration in the ancestor multiplied by 0.04. We also need access to the `time`-attribute of the `GenEnv` node.

GA_{19} is transported basipetally. The rule is similar to the previous one:

```
Organ [a:GA19] (? -ancestor-> Organ [b:GA19] ) ::> {
    double r = 0.01 * a[concentration] * DELTA_T;
    a[concentration] :-= r;
    if (b != null) {
        b[concentration] :+= r;
    }
}
```

But now the ancestor is put into an optional pattern `(? ...)` (see Sect. 6.5.7 on page 154). This means that even from the root, which has no ancestor, we have a basipetal transport to some imaginary organ. Put into other words, the root acts as a sink for GA_{19}.

Now on top of the two low levels of genetics and metabolism we specify the high level of barley morphogenesis. In principle, this specification consists of a set of rules which are the translation of an already existing pure L-system model of barley [16]. We do not give the details here, but show two exemplary rules for the combination of morphology with genetic and metabolic levels. The complete source code is part of the GroIMP distribution.

The influence of the metabolic level on morphogenesis is through GA_1 controlling internode elongation, see Fig. 10.10 on page 304. We model this

as a process which consumes GA_1 at a rate of its concentration multiplied by 0.1, and which increases the internode length by a rate proportional to the consumption rate:

```
i:Internode(rank) [s:GA1], (rank > 3) ::> {
    double r = 0.1 * s[concentration] * DELTA_T;
    s[concentration] :-= r;
    i[length] :+= 30 * r;
}
```

The influence of the genetic level on morphogenesis is both indirect and direct. The indirect influence is via the metabolic level as the dwarfing gene *Zeo* reduces the production of GA_{19} and, consequently, the production of GA_1, leading to a decreased rate of internode elongation. The direct influence is mostly of discrete nature, i. e., the presence or absence of genes switches between different parts of the right-hand sides of growth rules. As an example, if the *uniculm* gene is set to zero for both chromosome sets, there is no branching. We specify this directly in the growth rule for a `Meristem`, where the condition `((int)g[0][6] | (int)g[1][6]) != 0` tests for at least one gene set to one:

```
m:Meristem(pc, rank, order) (* [ga19:GA19] [ga20:GA20] [ga1:GA1]
    -ancestor-> Individual -env-> g:GenEnv *) ==>
    ...
    if ((((int)g[0][6] | (int)g[1][6]) != 0)
        && (rank <= 3) && (order < 3)) (
        [   RH(57) RL(40) // create branch, copy concentration values
            Meristem(1.5 * PLASTOCHRON, rank, order + 1)
                [cloneNode(ga19)] [cloneNode(ga20)] [cloneNode(ga1)]
        ]
    )
    ... m ...;
```

Figure 10.11 on the next page shows five full-grown barley individuals with their genome.

The presented barley breeder model has been extended to detect shade using information on the local distribution of radiation at different wavelengths [18]. For this purpose, the built-in radiation model of GroIMP is used (Sect. B.14 on page 412). Sensors attached to meristems measure the radiation in the red and far-red regions of the spectrum. This radiation is a result of the complex interaction of light, emitted by light sources, with geometry and optical properties of the growing barley individuals. In the extended model, the coefficients for reflection and transmission of leaves are set to higher values for the far-red spectrum compared to the red spectrum. Consequently, far-red light is less absorbed by the leaves, leading to a lower red:far-red ratio in shaded regions than in unshaded regions.

Plants typically react to shade by shade avoidance reactions, among them internode elongation, suppression of lateral branches and early flowering [18].

Figure 10.11. Five barley individuals with their genome. Uniculm (second from the left), dwarf (centre) and *cer-ze* (blue-green wax layer, centre and its neighbours) can easily be identified

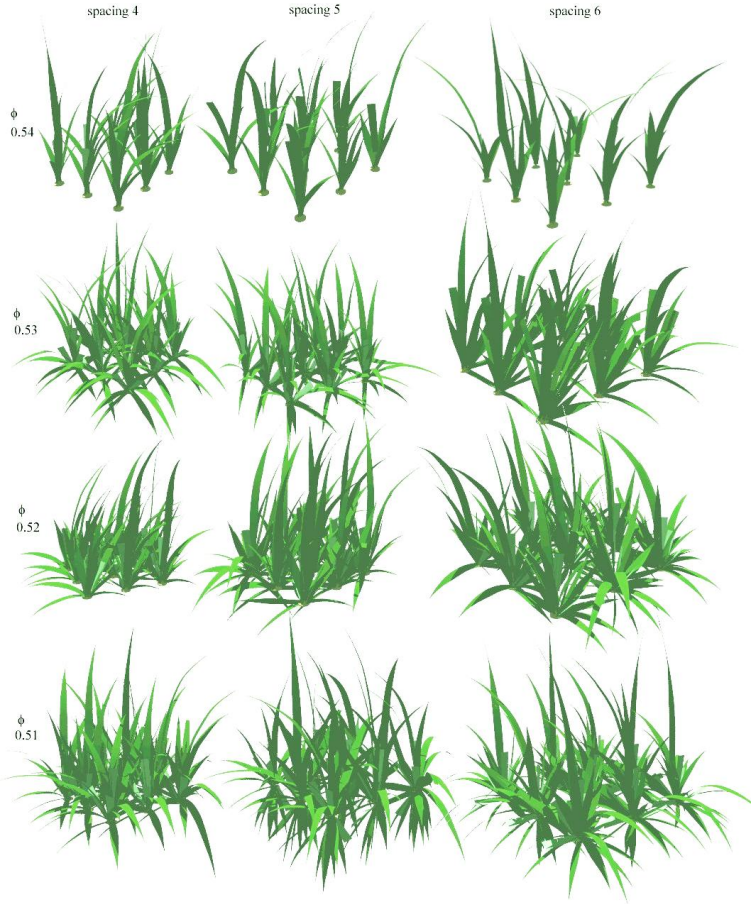


Figure 10.12. Simulated barley canopies at different settings for plant spacing and branching threshold ϕ (from [18]). Increased spacing and decreased threshold lead to increased branching

In our model, we make the production of GA_{19} dependent on the red:far-red ratio, and suppress branching if the ratio is below a given threshold. The result for different spacings and thresholds is visualized in Fig. 10.12 on the preceding page. More details about this extension can be found in [18].

The extension presented in [18] also contains a model for object avoidance during leaf growth. This was explained in more detail in [17]. The first prerequisite of such a detailed model of leaf growth is to abandon the representation of leaves by fixed, predefined surfaces (e. g., interactively modelled Bézier surfaces like in the basic version of the barley breeder) and, instead of this, to model the leaf as a chain of vertices along which a leaf profile curve is extruded. For this purpose, GroIMP provides several NURBS construction techniques [149]. We initialize such a leaf by code like

```
const Curve leafProfile = curve("leaf2");
...
Mark GrowingLeaf(size, 10, 0) NURBSSurface(leafProfile)
...
```

where the `NURBSSurface` node extrudes the interactively modelled curve referenced by `leafProfile` along the path defined by `Vertex` nodes between the nearest `Mark` ancestor and the surface. Initially, there are no vertices, but these are created by the growth rule for a `GrowingLeaf`:

```
gl:GrowingLeaf(size, n, l) ==>
    ...
    {de.grogra.vecmath.geom.Cone cc = cone(gl, false, 65.9);}
    if (empty((* f:F, ((distance(gl, f) < n + 20) && (f in cc)) *))) (
        // way is clear
        RV(0.5)
        Vertex(calcLeafSize(l / (300*size)))
        M(n)
        GrowingLeaf(size, min(n + 1, 10), n + 1)
    ) else ( // objects in the way, try to evade
        RU(random(-10, 10)) RL(random(-5, 5))
        Vertex(calcLeafSize(l / (300*size)))
        M(3)
        GrowingLeaf(size, 3, l + 3)
    )
    ... ;
```

The rule computes a one-sided cone `cc` with a half opening angle of 65.9 degrees. Its tip coincides with the location of the `GrowingLeaf` `gl`. (The computation is actually done by the method `cone` of `de.grogra.rgg.Library`.) If there is no `F` node (i. e., no internode) close to `gl` whose base point lies in `cc`, the way is clear, a new `Vertex` is inserted after an `RV`-node for a tendency to grow downwards (see Table B.2), a leaf growth of length `n` happens, and this growth length `n` is increased for the next step. Otherwise, a small random rotation is performed in order to try another, possibly better direction, a new `Vertex` is inserted, only a short growth of length 3 happens, and this length is also set to 3 for the next step. The factor in the constructor of `Vertex` scales

Figure 10.13. Enhanced barley model with detailed model of leaf shape and collision avoidance

the leaf profile, the used function `calcLeafSize` is such that the leaf surface starts and ends with small widths, but has a maximal width in the middle part. Figure 10.13 shows an individual barley grown with this mechanism of object avoidance.

Note that the growth rule of leaves only considers `F` nodes as obstacles, but nodes of leaves are not regarded. This could easily be changed by the inclusion of `Vertex` nodes in the condition for a clear way.

10.5.3 Carrot Field with Rodent

The next example is a simplistic model of a carrot field [100]. Individual carrots are modelled by a single cone-shaped root of type `Carrot` and a branched above-ground structure created by a single L-system rule (see [100] and the source code in the example gallery), internodes being represented by `F` nodes and leaves by the model-specific class `Leaf`. Already in the initialization the complete structure is set up, but all internodes have a length of zero. Their growth is governed by carbon that is produced in `Leaf`-objects and is transported downwards to the root. Carbon is represented by particles of class `Carbon` with a `value`-attribute for the amount of carbon. The stochastic production rule appends such a particle to each leaf with a probability of 5%:

```
x:Leaf ==>>
   if (probability(0.95)) (break)
   x [Carbon(0.03 / (1+0.8*first((* x -ancestor-> Carrot *)[shadow])))];
```

The carbon amount depends on the `shadow`-attribute of the carrot, which is a measure for the shading due to other carrot plants and will be discussed below. Now carbon particles are moved downwards along internodes:

```
n:. [c:Carbon] -ancestor-> a:F ==>> n, a[c];
```

Recall that the symbol `.` is a wildcard so that `n` matches nodes of any type.

For each carbon particle `c` at an internode `n`, there is a choice. Either the internode is the basal internode, i. e., it is directly connected with the `Carrot` root `m`, then we remove the particle and let the root grow. Otherwise, the internode consumes an amount of carbon and `n` elongates proportionally:

```
n:F [c:Carbon] ==>>
    {Carrot m = first((* n < Carrot *));}
    if (m != null) (
                // If n is the immediate successor of a carrot m
        n       // keep n in the graph, but delete c
                // and let the carrot grow (note that scale is negative)
        {m[scale] :-= 0.014 * c[value];}
    ) else {    // else allocate an amount v of carbon
        float v = 0.2 * c[value];
        c[value] :-= v;
                // and elongate the internode n.
        n[length] :+= v;
        break;  // Do not apply any structural changes to the graph.
    };
```

As we have already indicated, our model includes competition based on the `shadow`-attribute of carrots as a measure for mutual shading. For each carrot `c`, the value is taken to be the total length of all internodes `F` of other carrots `d` that have a base in `c`'s light cone, which has its centre at the centre point of `c`'s leaves, reduced to 30% in height, and a half opening angle of 50 degrees:

```
c:Carrot ::> { // Compute the centre of c's leaf origins
    Tuple3d m = mean(location((* c (-(branch|successor)->)* Leaf *)));
        // and reduce it along the z-axis.
    m.z *= 0.3;
        // The shadow-field will contain the total shadowing length.
    c[shadow] := sum // For every neighbouring d
        ((* d:Carrot, ((d != c) && (distance (c, d) < 3)),
                    // find all internode descendants f
            d ( -(branch|successor)-> )* f:F,
                    // within a light cone around m
            (f in cone(m, HEAD, 50))    // HEAD = (0, 0, 1): upwards
                    // and sum up their length.
        *)[length]);
}
```

This rule demonstrates how query expressions of the XL programming language and the 3D-algorithms of GroIMP like `distance`, `cone` and the overloading of the `in`-operator cooperate, allowing the modeller to implement both flexibly and concisely the competition for light. The modeller is not bound to a set of predefined functions that are provided as black boxes by the modelling software.

Until now, the carrot field model has a tree-like structure; it makes no use of true graphs. The structure changes if we include a water vole (*Arvicola terrestris*, a rodent) which feeds on carrot roots and digs a burrow system, thereby creating a true graph. The vole is represented either by an instance of `Vole` or by an instance of `DiggingVole`, depending on its state. The burrow system consists of nodes of type `Burrow`, connected with edges of type `net`. The vole and burrow nodes are represented by spheres, burrow connections (i. e., `net` edges) by cylinders using the instancing technique of the Sierpinski example (Sect. 10.1.2 on page 274). Burrow nodes have an attribute `scent` which is used as a scent mark: its value is increased when the vole is at the node, but there is also an exponential decay. This helps the vole to find out the recently visited burrow nodes. The rule for exponential decay is very simple:

```
b:Burrow ::> b[scent] :-= b[scent] * 0.3;
```

The next rule describes a non-digging vole that feeds on close carrot roots and moves along the existing burrow network:

```
b:Burrow [v:Vole] ==>>
    {   // reinforce scent mark
        b[scent] :+= 1;
        Carrot c = first((* x:Carrot, (distance(v, x) < 0.5) *));
        if (c != null) {       // there is a close carrot
            c[length] :*= 0.9; // gnaw off a bit
            c[topRadius] := 1 - 0.2*c[length];
            for ((* c -minDescendants-> r:RL *)) // and let the carrot
                r[angle] := 13 + (5-c[length]) * 6; // leaves sink down
        }
    }
    if ((c != null) // we have just gnawed, no need to dig a new tunnel
        || probability (0.8)) (
        {   // randomly select a neighbouring burrow n of b, preferring
            // burrow nodes with a less intensive scent mark
            Burrow next = selectRandomly((* b -net- n:Burrow *),
                                         Math.exp(-n[scent]));
            if (next == null) {
                next = b;
            }
        }
        b, next [v]
    ) else ( // no carrot found, try to find one by digging a new tunnel
        b RU(random(-90, 90)) M(1) DiggingVole
    );
```

The first part of this rule contains an important modification of the geometry of the above-ground part of carrots: if the vole gnaws off a bit from a close carrot root, the angles of all first `RL` rotations are modified such that the whole above-ground part sinks down. The path `c -minDescendants-> r:RL` is equivalent to `c (-(branch|successor)->)+ : (r:RL)`, but uses the more efficient method `minDescendants` of `de.grogra.rgg.Library`. By the competition model from above, the sinking down leads to an increased `shadow`-value of the affected carrots and, thus, to a decreased carbon production and a reduced elongation rate of internodes. The second part of the rule moves the vole to a neighbouring burrow node, or it changes the vole to a digging vole. This happens with a probability of 20% if the vole could not find a close carrot. It then randomly chooses a starting direction for digging and moves one unit in this direction by the turtle command `M`.

Being a digging vole, the animal shows a different behaviour. Given that it has not moved too far from the centre, it finds the closest burrow node `x` within a disc of radius 1 and in a cone in the movement direction with a half opening angle of 60 degrees. If such a node exists, the vole digs to this burrow node `x` and becomes a non-digging vole. Otherwise, it creates a new burrow node `n`, selects the closest carrot `c`, rotates towards this carrot in the manner of a positional tropism, but with an additional random rotation, and digs a distance of 0.5 in the new direction.

```
v:DiggingVole -ancestor-> b:Burrow ==>>
    if (ORIGIN.distance(v) > 5) (
        b [Vole] // The vole has moved too far: Stop digging.
    ) else (
        // Of all burrow nodes t close to v and within the forward cone
        // of the vole, select that burrow x with minimal distance to v.
        {Burrow x = selectWhereMin((* t:Burrow,
                                      ((t != b) && (distance(t, v) < 1)
                                       && (t in cone(v, false, 60))) *),
                                   distance(t, v));}
        if (x != null) (          // If such a burrow x exists, create
            b -net-> x, x [Vole] // a tunnel to it and stop digging.
        ) else ( // Otherwise, create a new burrow node n with tunnel,
            n:Burrow(1) [<-net- b]
                 // select the closest carrot c,
            {Carrot c = selectWhereMin((* d:Carrot *), distance(d, v));}
                 // and move v towards c.
            tropism(v, c, Math.exp(-0.1*distance(c, v)))
            RU(random(-10, 10)) M(0.5) v,
                 // Embed n at the graph location of v.
            moveIncoming(v, n, -1)
        )
    );
```

The used `cone`-method is another one as in the rule for shadow computation. The axis of the returned cone coincides with the local z-axis of the first argu-

ment `v`, i. e., with the movement direction of the vole (Sect. B.12 on page 406). The second **boolean** parameter is set to **false**, this controls whether the tip of the cone is given by the base or top point of `v`. As we represent a vole by a sphere, both points are the same (namely the centre of the sphere), but they might be different for other objects like cylinders. The last statement invokes the method `moveIncoming` on the current producer, which delegates the invocation to the underlying connection queue (Sect. 9.1.2 on page 237). On derivation, incoming edges of `v` are moved to `n` if they are of the type specified by the third argument. The provided argument `-1` is an abstract edge type which is a supertype of any edge type: recall that edge types are encoded by bitmasks, and we have $\gamma \,\&\, -1 = \gamma \neq 0$ for any edge type γ.

The preceding rules are put into a complete program that models a carrot field with a water vole. The invocation of the single rules is controlled by the application step in order to implement different time resolutions for the vole, the carrots and the relatively time-consuming shadow calculation:

```
if ((step % 25) == 0) [...] // shadow calculation
if ((step % 5) == 0) [...] // carrot growth
[...] // vole rules
step++;
```

Figure 10.14 on the facing page shows a snapshot of the model after a number of steps and with a user intervention.

10.5.4 Spruce Model of GROGRA

This example does not make use of the new features of GroIMP compared to GROGRA, it just demonstrates that source code in the language of GROGRA can also be used (Sect. B.15 on page 414). Figure 10.15 on the facing page shows the result of the L-system of a young spruce tree described in [106] at an age of 17 years.

10.5.5 Analysis of Structural Data of Beech Trees with XL

This example differs from the other ones as we do not use the XL programming language to specify growth or dynamics of a structure, but to analyse an existing plant structure. Such analysis is an important issue in functional-structural plant modelling, especially in the context of parameterization and validation. This holds equally for structures resulting from measurements of real plants and for modelled structures, i. e., the outcome of virtual plant simulations. The analysis has to consider both the topology of the structure and the values of parameters of its constitutive entities like geometry-related parameters or the internal state. For example, one may be interested in the number of internodes of growth units as a function of their age (a purely topological property) or in the branching angles, which also includes geometric information.

Figure 10.14. Snapshot of the carrot field model. The user has intervened by cutting off two leaves of the second carrot from the right; this favours its neighbours. However, the median carrot and its left neighbour suffer from being damaged by the vole

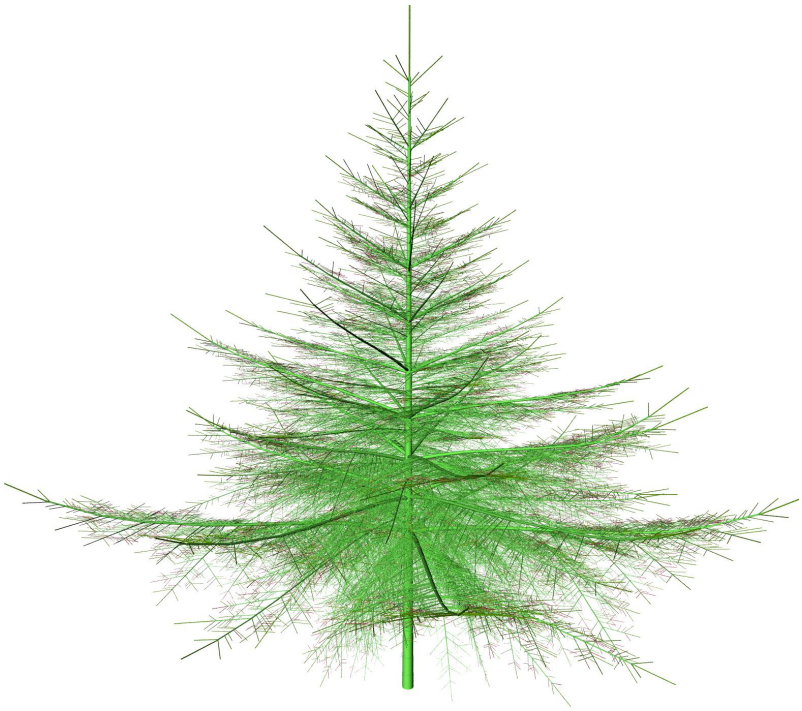


Figure 10.15. Spruce model of [106], executed and rendered within GroIMP

The AMAPmod software [68] is an example of a sophisticated program which has been specifically designed for plant structure analysis. Existing structures (the results of measurements or virtual plant simulations) are read in, the user then extracts the desired information by the querying language AML, and results can be visualized as a 3D-model or via several types of plots. But also the XL programming language can be used for analysis due to its graph query features. As an example, let us consider a tree represented by a structure of only `DTGShoot` nodes. This can be obtained within GroIMP by the import filter from DTD files which contain morphological descriptions of trees usually coming from measurements (Sect. A.6.2 on page 390 and [108]). Having such a structure, the expression

```
((* p:DTGShoot [b:DTGShoot], (p.order == 0) *),
 angle(direction(p), direction(b)))
```

yields the branching angles between all parent shoots `p` of order zero and their branching shoots `b`. The methods `angle` and `direction` are defined in the class `de.grogra.rgg.Library`. If we apply the aggregate method `statistics` of `de.grogra.rgg.Library` to this expression, we obtain a statistics of the branching angles including mean value and deviation.

For the beech model of the next section, we analyse data of young beech trees (*Fagus sylvatica* L.) which were measured in 1995 in the Solling (German midland mountains) [180, 76]. Their age ranges from 7 to 14 years. The complete topology was recorded with annual shoots being the elementary units, and for each shoot its length, number of internodes, branching angle and some further parameters were measured. Now we extract two simple properties out of the data. At first, the branching angle is analysed as explained above. Applied to the measured data, we obtain $64^\circ \pm 15^\circ$ for order zero and $50^\circ \pm 11^\circ$ for higher orders. As a second property, the relation between the length of a shoot and the number of its internodes is studied. For each shoot `a` which is not a short shoot (longer than 8 mm), the pair (`a.length`, `a.internodeCount`) is written to a `table`, where we use different rows of the table for different orders:

```
const DatasetRef table = dataset("Internode Count / Length");
...
for ((* a:DTGShoot, (a.length > 0.008) *)) {
    table.getRow(a.order).add(a.length, a.internodeCount);
}
```

This table can be exported to various formats like CSV or XLS if one wants to use external tools to further analyse the data. But it can also be plotted within GroIMP by

```
chart(table, SCATTER_PLOT);
```

Figure 10.16 shows the resulting plot for five imported beech trees. A linear relation $n(l) = p_0 l + p_1$ between length l and number n seems to be suitable.

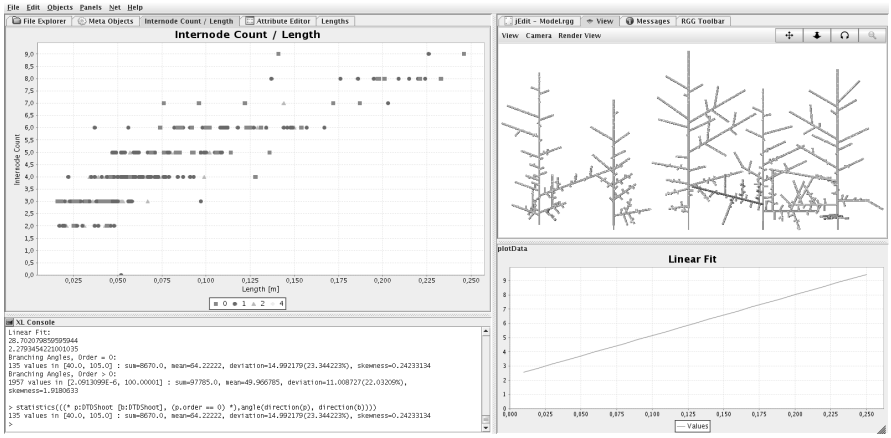


Figure 10.16. GroIMP window showing imported beech trees and result of analysis

By an anonymous function expression (Sect. 6.13.5 on page 188), we set up this relation as a `DoubleToDouble` function and specify initial values for p_0, p_1:

```
final double[] p = {25, 2};
DoubleToDouble n = double x => double p[0]*x + p[1];
```

The method `fitParameters` of `de.grogra.rgg.Library` can be used to compute the optimal values for p_0, p_1 (namely $0.287\,\mathrm{cm}^{-1}$ and 2.28, respectively):

```
fitParameters(n, table, p, new double[]{0.01, 0.0001});
```

The last argument to `fitParameters` defines the desired precision. A plot of the resulting function is also shown in Fig. 10.16, it is generated by the method `plot`:

```
plot(n, 0.01 * (1:25));
```

10.5.6 Beech Model and Tree Competition

This model is a relatively complex functional-structural model of young beech trees (*Fagus sylvatica* L.). It consists of several parts: The radiation model of GroIMP (Sect. B.14 on page 412) is used to compute the amount of absorbed light for each leaf. By a photosynthesis model, this amount is converted into carbon assimilates and allocated in the tree both by basipetal transport and global distribution, leading to secondary growth in girth or self-pruning if the supply is insufficient. From the availability of carbon assimilates we compute a vitality for buds, this then controls the number and lengths of created internodes in an annual step. At the end of this section, we will combine the model with an enhanced version of the spruce model from above to simulate a young mixed spruce-beech stand [76].

Entities of the Model

For the beech model, we use five specific classes for its entities. A `BeechLeaf` is a rectangular `Parallelogram` with a scanned image of a beech leaf as texture. The size of (the enclosing rectangle of) a new leaf is determined by two normally distributed random numbers with mean value 6 cm by 4.3 cm (note that we use coherent SI units throughout this section). A `BeechLeaf` has an attribute `producedCarbon` which stores the amount of produced carbon assimilates in a single step, the unit being mol C.

```
module BeechLeaf extends Parallelogram(normal(0.06, 0.015),
                                       normal(0.043, 0.01))
    .(setShader(leafShader)) {
    float producedCarbon;
}
```

The next class `Organ` is the superclass for internodes and the single root compartment. An organ has a cylindrical shape with an initial radius of almost zero. It will grow in girth as part of secondary growth, but the length will stay constant. The topological order (i. e., the stem has order 0, main branches have order 1 etc.) is stored in the attribute `order`. `allocatedCarbon` contains the amount of carbon allocated by the organ as part of the basipetal transport of carbon, `exportedCarbon` the exported part which is transported to the next organ in basipetal direction. `producedCarbon` is the total amount of carbon produced in all leaves above the organ. The attribute `preference` is a measure for the preference of the organ resulting from its carbon supply compared to other organs.

```
module Organ(super.length, int order) extends Cylinder(length, 0.0001) {
    float allocatedCarbon;
    float exportedCarbon;
    float producedCarbon;
    float preference = 1;
}
```

The single `Beech` node of the model represents a compartment which stands for all roots. This is a typical simplification of plant models as relatively less is known about the detailed below-ground growth of plants.

```
module Beech extends Organ(0.01, 0).(setShader(EGA_6));
```

Also `Internode` inherits from `Organ`, here we set the shader depending on the order.

```
module Internode(super.length, super.order) extends Organ
    .(setShader((order==0) ? stemShader:branchShader), setScaleV(true));
```

The effect of the invocation `setScaleV(true)` is used when the texture image for bark is mapped on an internode: by default, such an image is wrapped exactly once around the cylindrical surface. But as the internodes have different lengths, the global scale of the bark differs from internode to internode,

which is not desirable. By setting the `scaleV`-attribute to **true**, the global scale along the axes is the same for all internodes.

The final specific entity of the model is the `Bud`. Like an organ, it stores the topological `order`. The attribute `vitality` is a measure for the vitality of the bud and results from carbon production. `sign` alternates between -1 and 1 in order to implement alternate growth. `internode`, `leaf` and `beech` are references to the bearing internode, the leaf in whose axil the bud emerges, and the beech individual to which the leaf belongs, respectively.

```
module Bud(int order, float vitality, int sign, Internode internode,
           BeechLeaf leaf, Beech beech);
```

Primary Growth

Beech growth is implemented by a single L-system rule which specifies the annual growth of a bud into a shoot composed of several internodes with lateral and terminal leaves and buds. At first, we compute several parameters for the rest of the rule:

```
b:Bud(o, v, s,,, t) ==>
    {
        boolean createShort = v < 2.2;
        int count;
        float len;
        if (createShort) {
            count = 3;
            len = 0.01;
        } else {
            count = (int) v;
            if (probability(v - count)) {
                count++;
            }
            len = Math.max((count - 2.28) / 28.7, 0.02);
        }
        int budmin = count - Math.round(0.666f * (v - 0.875f));
        int sign = s;
        float[] lenDist = lengths(count);
    }
    ... ; // rest of growth rule, see below
```

`createShort` specifies if the bud shall give rise to a short shoot, i. e., to a shoot which does not elongate significantly and whose lateral buds normally do not grow out into shoots in the next year. This is assumed to be the case if the vitality `v` of the bud is below the threshold 2.2. `count` determines the number of internodes of the shoot. If the bud does not create a short shoot, it is set to a rounded value of `v` with a stochastic component such that the expectation value equals `v`. `len` specifies the total length of the shoot. For normal shoots, this is computed by the relation which was obtained from experimental data

in the previous section on page 317. If the internodes are numbered from 1 to `count`, all (non-terminal) internodes with a number greater than or equal to `budmin` will have a lateral bud. Like the relation between internode count and length, the formula for the relation between vitality and the number of buds was derived from the data of [180] (with the substitution of vitality by the internode count). `sign` is initialized with `s`, this will be multiplied by -1 for each internode to implement alternate growth. The last parameter `lenDist` is the normalized distribution of the total length among the individual internodes and was taken directly from an analysis done in [180].

Having these parameters, the growth rule continues by creating `count` internodes. Each internode bears a leaf. The terminal internode and some intermediate internodes also bear terminal and lateral buds, respectively.

```
1  b:Bud(o, v, s,,, t) ==>
2      ... // computation of parameters, see above
3      for (int i : 1 : count) (
4          {
5              sign = -sign;
6              boolean terminal = i == count;
7              float q = ((float) i / count) ** VIT_POWER_0;
8              float vit = Math.max(v*VIT_A*q / (1 + VIT_B*q), VIT_MIN);
9          }
10         x:Internode(lenDist[i-1] * len, o)
11
12         RU(sign * normal((o == 0) ? 0 : 15, (o == 0) ? 5 : 10))
13         RH(normal(0, (o == 0) ? 20 : 10))
14
15         if (terminal) (
16             if (o == 0) (RD(HEAD, 0.5))
17             [RL(90) RD(sun, 0.4) RL(-90) l:BeechLeaf]
18             Bud(o, vit, sign, x, l, t)
19         ) else (
20             [
21                 if (o == 0) (
22                     RL(sign * normal(64, 15))
23                 ) else (
24                     RU(-sign * normal(50, 11))
25                 )
26                 AdjustLU
27                 [RL(90) RD(sun, 0.4) RL(-90) l:BeechLeaf]
28                 if (!createShort && (i >= budmin)) (
29                     Bud(o + 1, vit, sign, x, l, t)
30                 )
31             ]
32         )
33     );
```

The rotation angles are guessed from the visual appearance with the exception of the branching angles in lines 22, 24 which were taken from the analysis of

the previous section (page 316). In order to orientate the leaves such that their surface receives a high amount of sunlight, we use a directional tropism `de.grogra.rgg.RD` towards the direction `sun`, which is a parameter of the model and set to $(-1, 0, 0.5)$ (lines 17, 27). But we have to enclose the tropism in a sequence `RL(90) RD(sun, 0.4) RL(-90)`, otherwise the growth direction (i. e., the longitudinal axis) and not the surface normal of the leaves would be rotated towards the sun.

The computation in lines 7, 8 has the vitality `vit` of bud `i` as its result. It is computed as the vitality `v` of the shooting bud `b`, multiplied by a factor between 0 and 1 which is a function of the fraction `i/count`. The parameters of this function have been chosen such that the vitality of the lower buds is nearly zero, while the terminal bud has the same vitality as the shooting bud.

Photosynthesis

Up to now, the model is a purely structural model. Now we integrate a functional component which computes carbon production and allocation and removes branches with insufficient supply. The starting point for this functional component is the inclusion of a photosynthesis model. This is driven by the interception of light in leaves. An accurate way to compute this interception is the built-in radiation model of GroIMP (Sect. B.14 on page 412) which takes into account the complete geometry and optical properties of plants and possible further objects.

```
const LightModel radiation = new LightModel(1000000, 6, 0.001);
```

This line initializes the radiation model with $1,000,000$ light rays per computation, a maximum ray depth of 6 and a threshold of 0.001 W for the power of light rays which shall be ignored. Furthermore, we have to assign suitable shaders to the leaves and the other plant organs (e. g., the reflection and transmission coefficients have to be set correctly) and to define one or more light sources. If this is done, the invocation

```
radiation.compute();
```

computes the amount of absorbed light for each node of the scene. This can be queried for a given node n by

```
radiation.getAbsorbedPower(n)
```

The result is an instance of the interface `de.grogra.ray.physics.Spectrum` which, in general, represents the spectral decomposition of some quantity. However, by the method `integrate` we can obtain the integrated value over the whole spectrum. We use this to define the method `produceCarbon` in the declaration of `BeechLeaf`:

```
public float produceCarbon() {
    float ppfd = radiation.getAbsorbedPower(this).integrate()
                 * (PPFD / AREA);
```

```
    float x = (EFFICIENCY * ppfd + Pmax) * (1 / (2 * M));
    float y = (EFFICIENCY * Pmax / M) * ppfd;
    float p = x - Math.sqrt(x * x - y);
    float c = AREA * Math.max(0, DURATION * p
     - SPECIFIC_MASS * C_FRACTION / (C_MASS * GROWTH_RESPIRATION_LEAF));
    this[producedCarbon] = c;
    return c;
}
```

The amount of absorbed light (measured in W) is divided by the average `AREA` of a leaf to compute the specific absorbed power. Then it is multiplied by `PPFD` to convert from $\mathrm{W\,m^{-2}}$ to the *photosynthetically active photon flux density*, measured in mol $\mathrm{CO_2\,m^{-2}\,s^{-1}}$. The latter is generally used in biology due to the direct stoichiometric relation 8:1 between absorbed photons and the binding of $\mathrm{CO_2}$ molecules. Based on an average wavelength of $\lambda = 550\,\mathrm{nm}$, we have $\texttt{PPFD} = \frac{\lambda}{8hcN_A} \approx 5.75 \times 10^{-7}\ \mathrm{mol\,CO_2\,J^{-1}}$. In principle, we also have to take into account the spectral composition of light as the photosynthetically active part of the spectrum ranges only from wavelengths between 400 and 700 nm, but as the whole model is rather a sophisticated toy model than a precise validated growth model, we neglect this fact. The computed `ppfd` is not directly the amount of produced carbon as the efficiency is far from 100%. In general, there is a linear relationship (with efficiency factor α) at low irradiance values and a maximum carbon production rate P_{max} in the light-saturated case. The above code uses the formula

$$P = \frac{\alpha i + P_{max}}{2M} - \sqrt{\left(\frac{\alpha i + P_{max}}{2M}\right)^2 - \frac{\alpha i P_{max}}{M}} \tag{10.1}$$

of [150] to compute the assimilation rate with $i = \texttt{ppfd}, \alpha = \texttt{EFFICIENCY} = 0.2, P_{max} = 1.4 \times 10^{-4}\,\mathrm{mol\,CO_2\,m^{-2}\,s^{-1}}, M = 0.98$ [150, 134]. This rate is multiplied by $\texttt{DURATION} = 4.32 \times 10^6\,\mathrm{s}$ to compute the produced carbon per leaf area for the whole year. Assuming an active leaf lifetime of 120 days, this corresponds to 10 hours of full light per day. From the produced carbon per leaf area, we subtract the amount required for growth of the leaves themselves. The latter results from the specific dry mass of beech leaves ($\texttt{SPECIFIC\_MASS} = 35\,\mathrm{g\,m^{-2}}$), the contribution of carbon to dry mass ($\texttt{C\_FRACTION} = 0.48$), the molar weight of carbon ($\texttt{C\_MASS} = 12.0107\,\mathrm{g\,mol^{-1}}$), and the ratio between carbon growth and available carbon ($\texttt{GROWTH\_RESPIRATION\_LEAF} = 1/1.2$ [39], the difference is needed for growth respiration). Finally, we multiply by `AREA` to obtain the produced carbon for the single leaf. This carbon is available for the rest of the plant and remembered in the attribute `producedCarbon` of the leaf.

Allocation and Transport

The produced carbon is distributed in the plant by an allocation model. We assume a basipetal transport of carbon. Each organ receives an amount

`imported` of imported carbon from its organ children and an amount `produced` of produced carbon from the leaf which it bears, if any. The carbon is allocated in a method `transportCarbon` of `Organ`:

```
void transportCarbon(float imported, float produced, float above) {
    imported += produced;
    float q = radius / 0.0025;
    float ex = imported * Math.exp(-0.7 * length * (1 + q) / q);
    this[allocatedCarbon] = imported - ex;
    this[exportedCarbon] = ex;
    this[producedCarbon] = above + produced;
}
```

At first, `produced` is added to `imported` in order to compute the total amount of carbon transported into the organ. A fraction `ex` thereof is exported downwards. Assuming that a fixed fraction κ of transported carbon is allocated per unit length of passed through tissue, the exported fraction for an organ of length l is given by an exponential law $e^{-\kappa l}$. The implementation makes κ dependent on the radius of the organ so that the allocated fraction is higher for thin organs (about 2.5 mm and below). This has been chosen solely by a manual optimization with respect to the visual appearance. The additional parameter `above` is the total amount of produced carbon of all leaf descendants which are not direct children. In the last line, we add the contribution by direct leaf children and store the sum in `producedCarbon`.

As the model executes a single step for a whole year, the complete transport from the leaves to the root has to be performed in that single step. Thus, `transportCarbon` has to be invoked on all organs. For a basipetal transport within a single step, all children of an organ have to be processed before the organ is processed itself, so that we may not choose an arbitrary order of the organs, but have to use a postorder traversal (see also Sect. 9.3.1 on page 263 with a similar problem for the pipe model). Specifically for this purpose, the pattern `de.grogra.rgg.basipetal` is defined. It is used as a path pattern and performs a postorder traversal along branch or successor edges starting at the in-parameter. For our model we write:

```
Beech -basipetal(Organ.class, children,
                 BeechLeaf.class, leaves)-> x:Organ ::>
    x.transportCarbon(sum(children[:][exportedCarbon]),
                      sum(leaves[:].produceCarbon()),
                      sum(children[:][producedCarbon]));
```

`x` is successively bound to all organs which are descendants of the found `Beech` node. This binding is in postorder traversal. The four parameters of `basipetal` in parentheses have the following meaning. `Organ.class` defines the type of nodes which shall be traversed. `children` is then of type `VoidToObjectGenerator<Organ>` and is a generator which yields all children of the current organ `x` when the operator `[:]` is applied to it (Sect. 6.3.3 on page 135). These two parameters suffice for the specification of a postorder

traversal, but we need two additional parameters as we are additionally interested in the leaves belonging to an organ, i. e., those leaves which can be reached from an organ without passing another organ (for our beech model, there is at most one such leaf). These parameters are similar to the first two parameters: `BeechLeaf.class` specifies the type of leaf nodes, `leaves` holds the found nodes for each organ `x`. The right-hand side of the rule invokes the method `transportCarbon` from above. It uses the photosynthesis model (method `produceCarbon`) to compute the amount of carbon produced in the `leaves`.

The carbon which is exported from the `Beech` node `b` of an individual tree cannot be transported downwards as such a node is the root node of the tree. Therefore, a fraction of this carbon amount is distributed over the whole plant, while the rest is allocated at the root node:

```
float distributed = 0.7 * b[exportedCarbon];
b[allocatedCarbon] += b[exportedCarbon] - distributed;
```

The distribution uses a weight which is computed for each organ by the method `distributionWeight` based on the demand of the organ due to maintenance respiration, the amount of allocated carbon, the `preference` attribute and the order. The details can be found in the source code in the example gallery of GroIMP. The resulting amount of distributed carbon for an individual organ is passed to the method `grow` of the organ:

```
float f = distributed
          / sum((* b -descendants-> Organ *).distributionWeight());
(* b -descendants-> o:Organ *).grow(f * o.distributionWeight());
```

Here, we do not have to take care of the order of traversal and use a simple query to find all organ descendants of the beech `b`. The method `grow` of class `Organ` implements secondary growth (growth in girth) with the sum of allocated and distributed carbon, reduced by an amount for maintenance respiration, as carbon input:

```
void grow(float distributedCarbon) {
    float input = allocatedCarbon + distributedCarbon
                  - maintenanceRespiration();
    if (input >= 0) {
        float m = input * (GROWTH_RESPIRATION_WOOD * C_MASS
                           / C_FRACTION);
        this[radius] =
            Math.sqrt(this[radius]**2 + m / (DENSITY*Math.PI*length));
    }
    this[mark] = (order > 0) && (input < 0);
}
```

The computation converts from the amount of carbon, multiplied by a factor `GROWTH_RESPIRATION_WOOD` $= 1/1.38$ [39] to account for growth respiration, to the resulting biomass `m` and, ultimately, to the corresponding new radius.

The last line sets a mark on non-stem organs having a negative input. Their supply by carbon is insufficient so that they fall off as a result of self-pruning. The same happens for all leaves as beech trees lose their leaves in autumn:

```
o:Organ & (o[mark]) ==>> ;
BeechLeaf ==>> ;
```

Maintenance respiration is assumed to be proportional to the surface of organs with an annual mean factor `MAINTENANCE_RESPIRATION` $= 1.92 \times 10^{-7}\,\mathrm{mol\,CO_2\,m^{-2}\,s^{-1}}$ (derived from [182]).

```
float maintenanceRespiration() {
    return 3600 * 24 * 365 * MAINTENANCE_RESPIRATION * 2 * Math.PI
        * radius * length;
}
```

In principle, the model is now complete. However, we have added an ad-hoc mechanism which modifies the `preference` attribute of organs and the `vitality` attribute of buds by an acropetal propagation, weighted by produced carbon at branching points. This implements a reaction of the plant to favour those parts with a better carbon supply, i. e., with less shading. The source code can be found in the example gallery.

Results

Figure 10.17. Beech individuals with different light conditions after ten years. From left to right: $50\,\mathrm{W\,m^{-2}}$, $100\,\mathrm{W\,m^{-2}}$, $200\,\mathrm{W\,m^{-2}}$

Figure 10.17 shows three outcomes of the simulation after ten years with different light conditions. The effect of light is in evidence. But this is of course only a very coarse measure of the plausibility of the model, one has to compare the results in more details and dimensions with experimental data. The analysis of results can be done with expressions of the XL programming language just like the analysis of experimental data (Sect. 10.5.5 on page 314).

For example, we may study the distribution of annual shoot lengths (which is an emergent property of the beech model) and compare it with the measured data obtained in Sect. 10.5.5 on page 314. Corresponding histograms are shown in Fig. 10.18, where again measurements from [180] were used. The distribution of simulated shoot lengths differs in shape from the distribution of measured shoot lengths. This could be the starting point for refinements of the model.

The current state of the model should be understood as a "proof of concept": the model itself is not meant to be realistic in the sense that it could be used to precisely predict beech growth, but it shows that the XL programming language, combined with the environment GroIMP, is in principle suitable for the implementation and analysis of sophisticated functional-structural plant models.

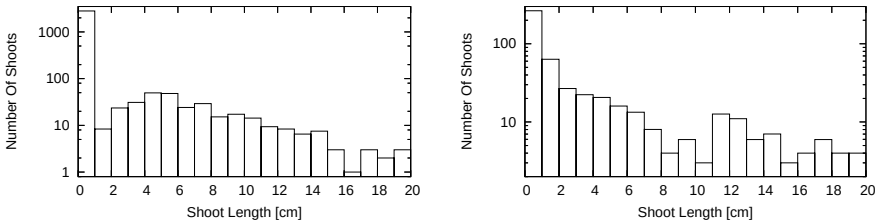
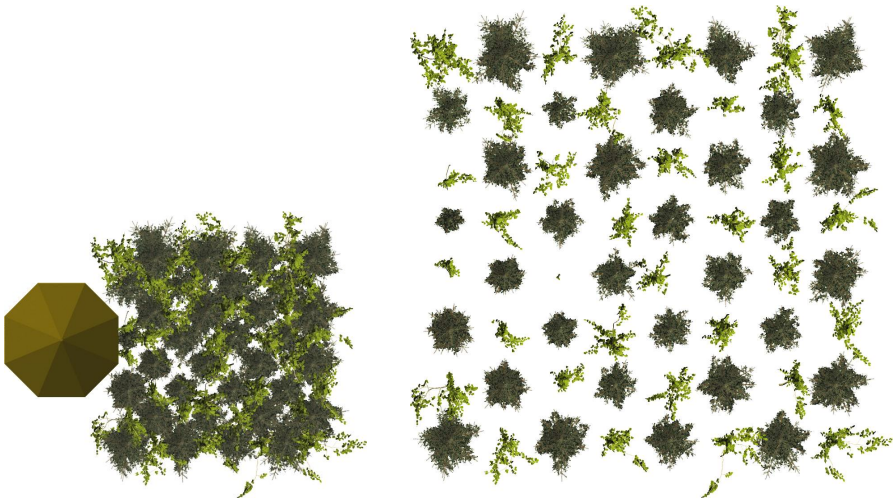


Figure 10.18. Distribution of shoot lengths: (**a**) measured data; (**b**) simulated data

Mixed Spruce-Beech Stand

The beech model was combined with an enhanced version of the model for young spruce trees of [106] (see also Sect. 10.5.4 on page 314). This enhanced version takes the received light by spruce needles into account, but not in the manner of the beech model. It rather uses the amount of received light as a limiting modifier for the otherwise context-free L-system-like growth. By combining both models and seeding several individuals of each species, one gets an interesting simulation of a mixed spruce-beech stand [76]. Figure 10.19 on the next page shows the result of a chessboard-like arrangement of 8 by 8 trees with an additional pavilion. The number of light rays for the radiation model had to be increased to 30 million to ensure the required accuracy. The simulation correctly shows the effects of shading: in the shade of the pavilion and neighbouring trees, growth is reduced. Figure 10.20 on page 328 uses the same model, but about 700 trees with a more irregular placement on a hilly landscape with a river.

(a)



(b) (c)

Figure 10.19. Simulation of spruce-beech model with direct sunlight coming from the left and diffuse light from the sky: (**a**) view on stand with pavilion after 11 years; (**b**) orthographic view from above; (**c**) exploded view, the shading effect of the pavilion and neighbouring trees can easily be seen

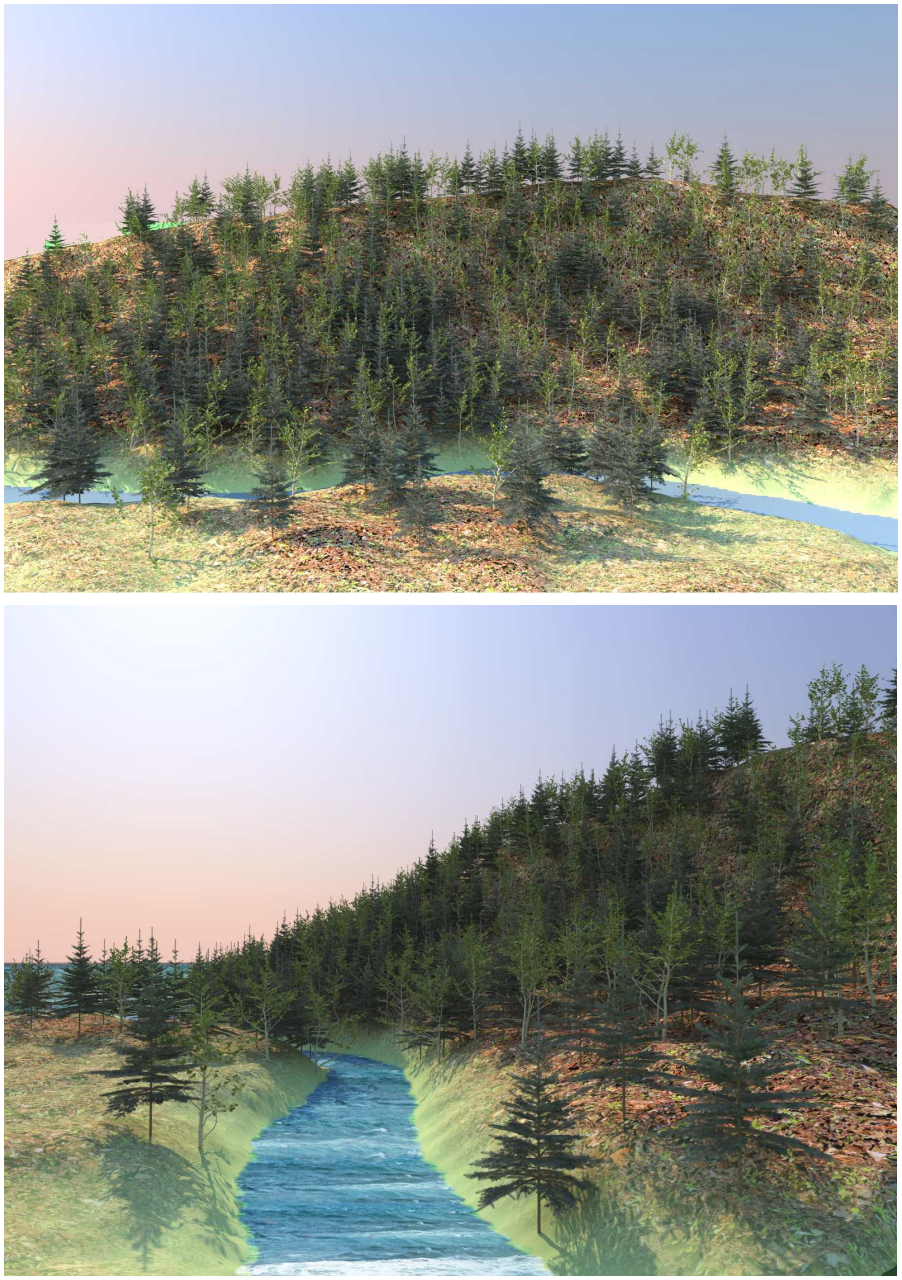


Figure 10.20. About 700 trees grown for 11 years. The final scene contained more than 11 million visible objects. It was computed on a 64-bit machine with 32 GiB RAM, an account for which was kindly provided by Andreas Hotho and Jörn Dreyer from the Knowledge and Data Engineering Group of the University of Kassel

10.5.7 Canola Model for Yield Optimization

This model is the result of a Diploma thesis [73]. The task was to implement a functional-structural model of canola (*Brassica napus* L.), including morphogenesis, nitrogen budget and carbon budget. The main purpose was to use this model to carry out virtual experiments for the optimization of yield with respect to the amount and time of nitrogen fertilization. The practical use of such an optimization is the reduction of nitrogen gift in order to both increase the efficiency of crop growing and decrease the environmental damage.

The developed model contains detailed submodels for physiological processes like senescence, respiration, remobilization and photosynthesis. Environmental parameters like temperature, humidity, day length, cloudiness and, most important for this study, nitrogen gift during the growth period are taken into account. Two optimization algorithms (hill climbing and threshold accepting) are used to find optimal fertilization strategies for a given morphology, or, with fixed fertilization strategy, to find an optimal morphology (which in practice would have to be achieved by breeding).

Figure 10.21. Visualization of the canola model after 70 and 80 days, respectively (from [73])

The results obtained with the model were consistent with empirical data and, concerning the computed optimal fertilization strategy, with agronomic recommendations. Thus, the model could be used as a good starting point for a refined version with increased prediction capabilities, but this was beyond the scope of the Diploma thesis. Figure 10.21 on the previous page shows the outcome of the model.

10.5.8 GroIMP as HTTP Server in an E-Learning Project

This example presents a system which was developed as part of a project within the "eLearning Academic Network Niedersachsen" [113]. The system allows students to explore virtual forest stands, make decisions on the fate of individual trees, and see a prognosis according to a growth model on the basis of their decisions. Thus, the aim is to have an interactive, collaborative e-learning system to teach models of forest growth.

The structure of the system is shown in Fig. 10.22 on the facing page. Several Virtual Forester clients connect to a central Elan Sim server which provides the clients with a scene description encoded in the VRML language [82]. Each client has a graphical 3D user interface in which the scene can be explored interactively. One may mark or fell trees, which is communicated to the server and from this to all other clients by events. When the exploration and manipulation of the current forest stand is complete, the Elan Sim server sends this current stand, again encoded as a VRML scene, to a growth engine via the POST method of the Hypertext Transfer Protocol (HTTP, [57]). The task of the growth engine is to compute a prognosis of the forest stand in a given number of years based on a growth model and the current stand. After completion of the computation, the Elan Sim server fetches the prognosis as a new VRML scene from the growth engine via the GET method of HTTP. The clients then may opt to switch to the prognosis in order to investigate the effects of the previously made decisions.

The interface between the Elan Sim server and the growth engine allows a wide variety of growth engines. The main requirement is that the growth engine can be addressed as an HTTP server by the HTTP protocol. For GroIMP to be used as a growth engine for the Elan Sim server, we implemented a simple HTTP server component which parses HTTP requests and passes them to GroIMP projects which are addressed by the URL part of the request.

Such a project was developed for conifer stands as part of a doctoral thesis [113]. At first, it scans the incoming VRML scene and extracts the necessary data. Each tree is represented by its position, height, diameter at breast height, crown height and diameter, age, biomass and species. The trees are used as input to the growth model of [148]. It incorporates photosynthesis based on a simplified crown representation, respiration and allocation, and it computes a prognosis which is put to a VRML file. This may be accessed by later HTTP GET requests of the Elan Sim server.

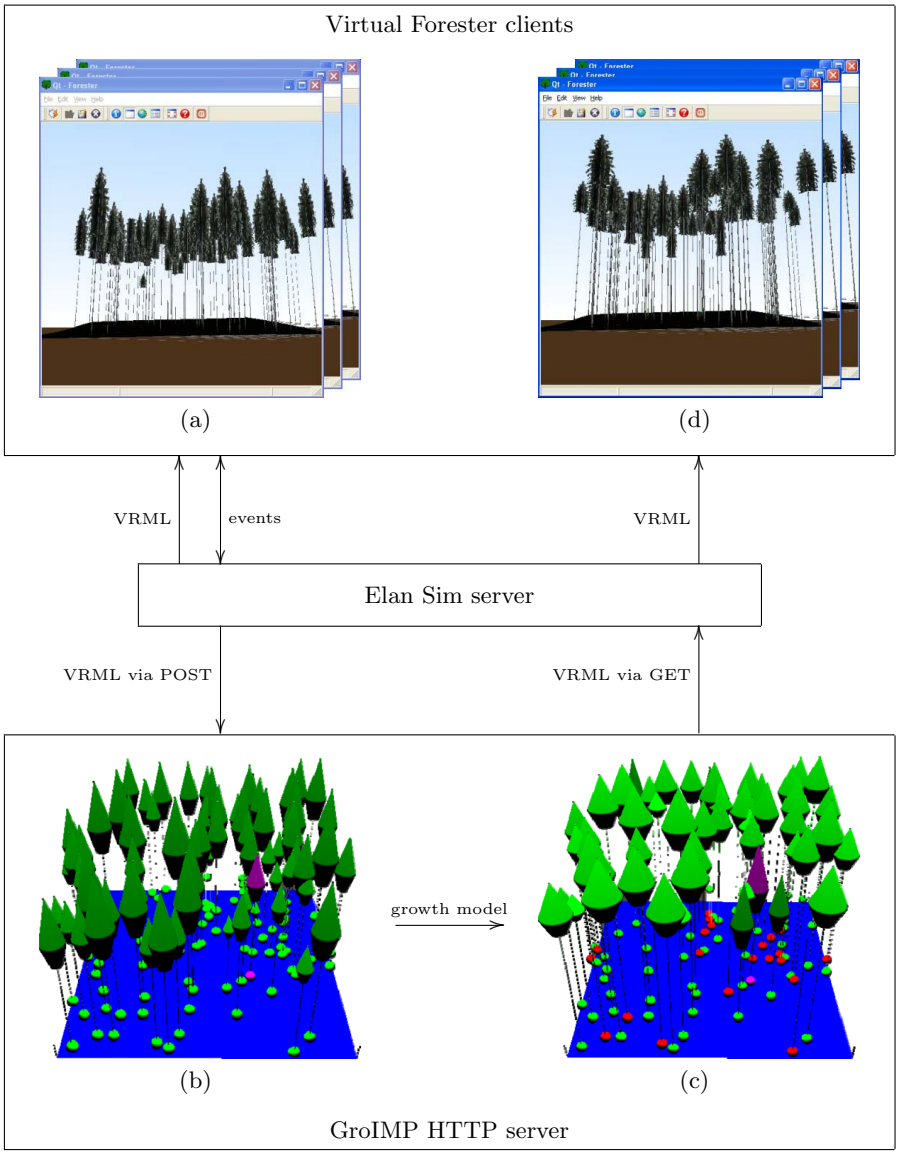


Figure 10.22. Usage of GroIMP as growth generator for the Elan Sim server: The Elan Sim server sends a VRML scene description to all connected Virtual Forester clients (**a**). Interactions made within the clients like marking or felling of trees are communicated by events through the Elan Sim server. After a complete thinning, the VRML scene description is sent to the GroIMP HTTP server by the POST method of HTTP (**b**). This starts a growth generator, implemented by a relational growth grammar, and its result (**c**) is obtained as a new VRML scene by the Elan Sim server via the GET method of HTTP. It may be passed on to the Virtual Forester clients (**d**). Screenshots taken from [113]

10.5.9 Reproducing an Alder Tree of the Branitzer Park

This model is the result of a bachelor thesis [165]. The task was issued in the framework of a larger project "Virtueller Branitzer Park" where a 3D model of the famous Branitzer Park in Cottbus, consisting of several buildings and about 8000 trees, is developed for display in VRML browsers. Within the thesis, several techniques were studied to create 3D models of trees, among them the rule-based technique on the basis of the XL programming language and GroIMP. It was possible to roughly reconstruct the shape of a specific given alder tree (*Alnus glutinosa*), see Fig. 10.23(a) and (b) on the next page.

The result of such a detailed rule-based model is a very complex geometry. For an interactive visualization of the whole park of 8000 trees, the geometry has to be simplified substantially. For trees very distant from the viewer, a common technique is to use a simple *billboard*, i. e., a single rectangle with a rendered view of the tree as texture and which is rotated such that it faces the viewer. However, when the viewer comes close to such a billboard, the simplification becomes noticeable. *Dynamic billboards* solve this problem by using several views from different perspectives. Usually, n lateral views are rendered by rotating the camera around the object and dividing the circle into n equal sectors. On display, we choose the view for the sector in which the viewer is located.

Figure 10.23(c) on the facing page shows such a dynamic billboard for the alder tree. The 72 lateral view were computed by the integrated raytracer Twilight of GroIMP (Sect. A.7.1 on page 392). The camera placement was done automatically by an algorithm implemented as part of the bachelor thesis. This also supports some further billboard techniques.

10.5.10 Ivy Model

The ivy model uses a mechanism provided by the class `AvoidIntersection` which allows to (approximately) move at a given distance along the surface of an object, thus simulating the behaviour of a climbing plant [179]. The mechanism shoots a number of test rays in order to scan the neighbourhood for obstacles or for the surface to follow, and computes a rotation so that the movement direction is adjusted accordingly. The result is shown in Fig. 10.24 on page 334 with a wall and an invisible sphere as surfaces to follow. Models for climbing plants were also implemented with open L-systems [129].

Figure 10.25 on page 335 shows a nice picture where the ivy model was combined with the spruce and alder models from above and models for grass, daisy and fern.

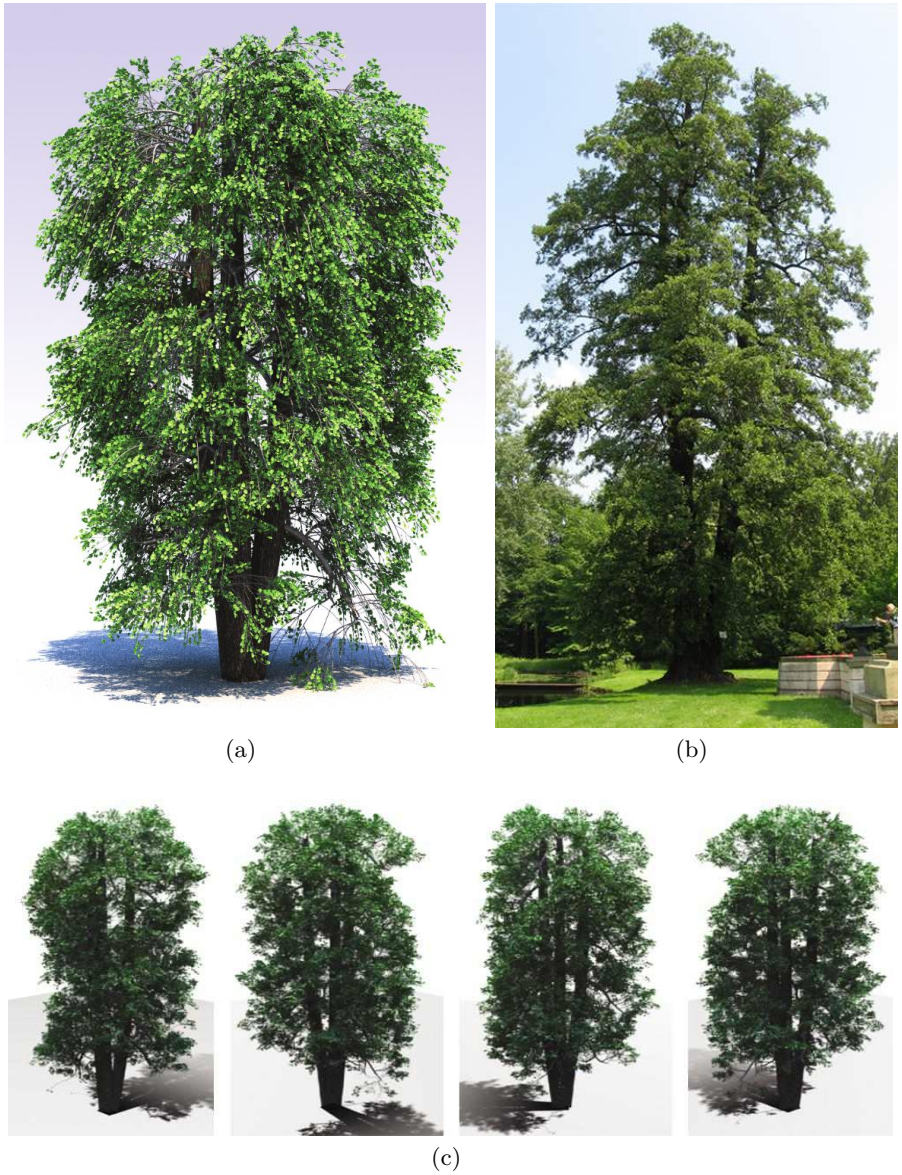
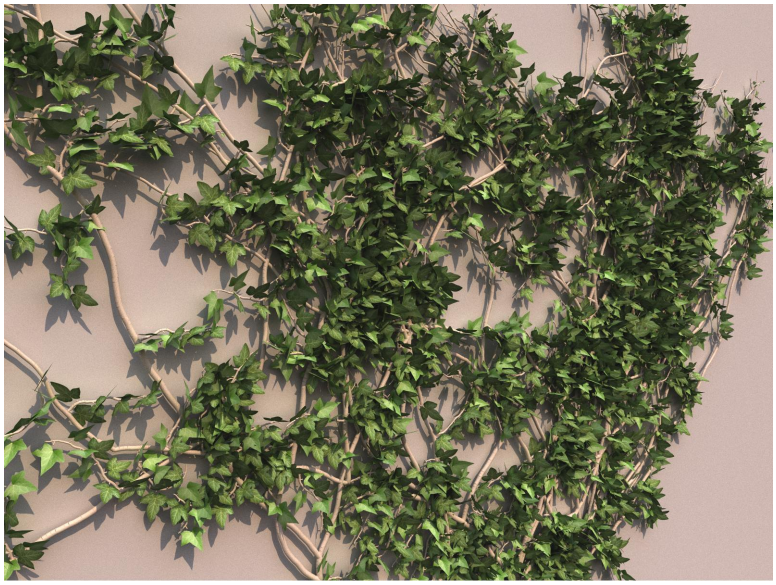
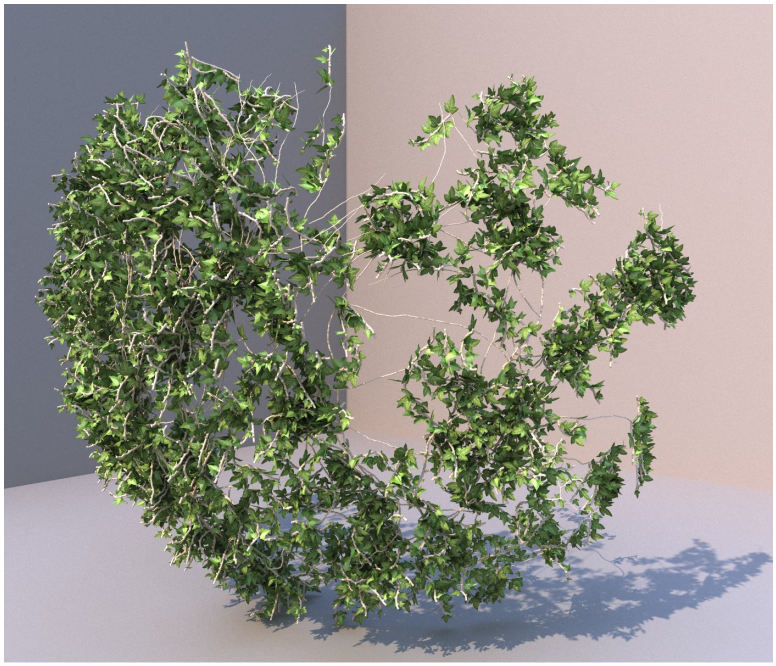


Figure 10.23. Alder tree (from [165]): (**a**) rendered view of tree; (**b**) photograph of original tree in Branitzer Park; (**c**) dynamic billboard with 72 sectors, displayed in a VRML browser



(a)



(b)

Figure 10.24. Ivy model (from [179]): (**a**) growing on a wall; (**b**) growing on an (invisible) sphere

Figure 10.25. Combination of ivy model with alder and spruce models from above and further models for fern, daisy and grass (from [179])

10.6 Graph Rotation Systems and the Vertex-Vertex Algebra

A very interesting usage of the XL programming language is the specification of a parallel variant of the vertex-vertex algebra. This algebra is the underlying formalism of the vv system (see Sect. 4.8.4 on page 85 and [178, 177]) which allows the programmed manipulation of graph rotation systems, a special representation of polygon meshes where each vertex stores a circular list of its neighbours. Table 10.1 shows the editing operations of the vertex-vertex algebra and their syntax in the vv system. Some of the operations were already presented in Sect. 4.8.4.

In the original approach, the editing operations of the algebra were applied immediately to the mesh. Due to the sequential mode of execution, this leads to several problems concerning the order of the individual operations as different orders may lead to different results. As a solution, the vv system provides the **synchronize** statement (page 87) which makes a copy of the current state and allows subsequent operations to refer to this static copy while modifying the mesh.

From the implementation of parallel derivations with the help of modifications queues (Sect. 9.1.2 on page 237) we know another mechanism to elimi-

Description	vv statement	XL statement
set neighbourhood of v to circular list (a, b, c)	**make** {a, b, c} **nb_of** v	v [a b c]
erase x from neighbourhood of v	**erase** x **from** v	~x **in** v
substitute x for a in neighbourhood of v	**replace** a **with** x **in** v	a >> x **in** v
insert x after a in neighbourhood of v	**splice** x **after** a **in** v	a x **in** v
insert x before a in neighbourhood of v	**splice** x **before** a **in** v	x a **in** v
split edge from a to b by insertion of x	-	a <+ x +> b

Table 10.1. Editing operations of the vertex-vertex algebra and their notation in the vv system and the XL programming language when `VVProducer` is used

nate the dependence on execution order. This mechanism can easily be utilized for the implementation of a true parallel version of the vertex-vertex algebra by defining an own modification queue for the operations of Table 10.1. Furthermore, the syntax of production statements (Sect. 6.7 on page 162) allows a convenient syntax for these operations if we provide a suitable specialized producer. The producer `de.grogra.rgg.model.VVProducer` was designed for this purpose and provides all operations as shown in Table 10.1. Each such statement leads to a corresponding entry in the vertex-vertex modification queue which is executed on invocation of the method `derive` together with all other entries of this and the other modification queues.

The chosen mapping to XL statements raises the interesting question how to prevent undefined operations already at compile-time. For example, the user may write `a x in v` or `~x in v`, but not `~a x in v` or `x in v`. In Sect. 6.7.3 on page 167 we discussed how the mechanism of production statements, where each statement returns the producer for the next statement, can be used to specify a deterministic finite automaton where states correspond to producer types and transitions to operator methods of the types. By doing its usual job, the compiler checks if the input is accepted by the automaton. For the syntax of Table 10.1, we have the automaton of Fig. 10.26 on the facing page.

As example for the translation of the automaton to producer types and operator methods, consider the state V. The corresponding producer type is the class `InsertReplaceSplitSet` nested in `VVProducer` with the methods shown in the following class diagram:

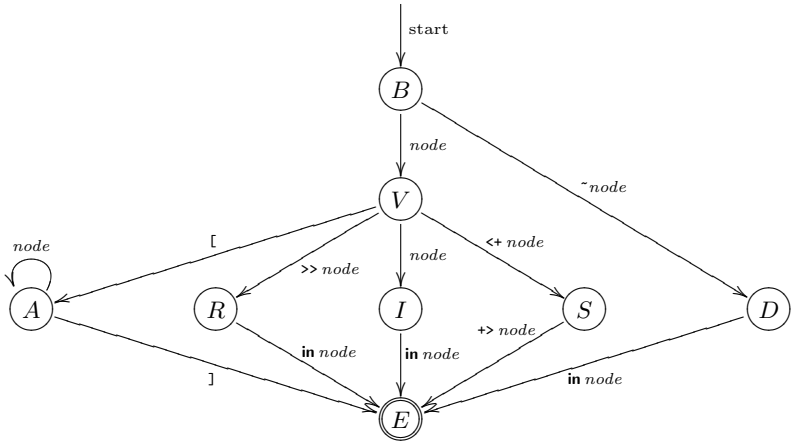


Figure 10.26. Deterministic finite automaton for the syntax of the XL statements of Table 10.1. *node* stands for an arbitrary vertex-valued node expression

InsertReplaceSplitSet
+ operator$space(v: GRSVertex): InsertReplaceDelete
+ operator$shr(v: GRSVertex): InsertReplaceDelete
+ operator$plusLeftArrow(v: GRSVertex): Split
+ producer$push(): AddNeighbor
+ producer$pop(p: AddNeighbor): End

The producer class `InsertReplaceDelete` is used for the states I, R, D where a final **in** *node* is expected. `Split` corresponds to S, `AddNeighbor` to A and `End` to E.

The mechanism of `VVProducer` operates on the implementation of graph rotation systems as provided by GroIMP. There, a `MeshNode` has to be configured with a `GRSMesh` and has to bear all vertices of class `GRSVertex` as direct children (i. e., for each vertex there is an edge from the mesh node to the vertex). A `GRSVertex` has the list of neighbouring vertices as attribute. This list is the target of operations of the `VVProducer`. The class `GRSVertex` defines several useful operations on the list of neighbours which correspond to selection operations of the vv system:

GRSVertex
+ getNeighbors(): ObjectList⟨GRSVertex⟩
+ nextTo(v: GRSVertex): GRSVertex
+ prevTo(v: GRSVertex): GRSVertex
+ next(v: GRSVertex): GRSVertex
+ prev(v: GRSVertex): GRSVertex
+ first(): GRSVertex
...

The methods `nextTo` and `prevTo` return the vertex which follows or precedes, respectively, the vertex `v` in the list of neighbours of the current vertex. `next` and `prev` return the vertex which follows or precedes, respectively, the current vertex in the list of neighbours of `v` (i. e., `a.nextTo(b)` is equivalent to `b.next(a)`).

With the presented implementation of a parallel vertex-vertex algebra, the Loop subdivision scheme for triangular polygon meshes can be specified in a very similar way as for the original vv system. The original code is shown in Sect. 4.8.4 on page 86, its translation to the XL programming language looks as follows:

```
1  p:GRSVertex ::> {
2      int n = p.valence();
3      double w = 0.625 - (0.375 + 0.25 * cos(PI*2/n))**2;
4      Vector3d x = p[position] * (1-w);
5      for ((* p -neighbors-> q:. *)) {
6          x += (w / n) * q[position];
7          if (p < q) {
8              Vector3d t = 0.375 * p[position];
9              t += 0.375 * q[position];
10             t += 0.125 * (p.nextTo(q))[position];
11             t += 0.125 * (p.prevTo(q))[position];
12             ==> vv p <+ GRSVertex(t) +> q;
13         }
14     }
15     p[position] := x;
16 }
17
18 {derive();}
19
20 x:newGRSVertices -first-> p:GRSVertex -next(x)-> q:GRSVertex
21     ==>> vv x [x.next(q) q x.prev(q) x.next(p) p x.prev(p)];
```

The first rule splits each edge, i. e., each unordered pair (p, q) of neighbouring vertices by the insertion of a new vertex. The splitting operation is specified in line 12 where we make use of a stand-alone production statement (Sect. 6.9 on page 174). Its producer is implicitly given by the surrounding execution rule, it is the normal `RGGProducer` of GroIMP (Sect. B.8 on page 401). By the specification of `vv`, we switch to the `VVProducer`. Internally, this is achieved by a simple trick: the class `RGGProducer` contains the definitions

```
public VVProducer vv() {
    ... // return corresponding VVProducer
}

public VVProducer operator$space(VVProducer prod) {
    return prod;
}
```

I. e., `vv` returns the `VVProducer` associated with the `RGGProducer`, and, by the second method, this may be used as a non-prefixed node expression in production statements handled by the latter producer. However, such usage does not lead to any modification or side effect; by the return value of `operator$space` it simply switches to the `VVProducer` so that the following production statement is handled by this producer. The same trick is also used in line 21 where the neighbourhood of all new vertices is completed. The used method `newGRSVertices` in line 20 is defined in `de.grogra.rgg.Library` and yields all new vertices which were added by the vertex-vertex modification queue on derivation.

The main difference between our implementation and the original one lies in the temporal coordination of the application of changes. The original implementation initially creates a copy of the mesh by the **synchronize** statement, sequentially modifies the current mesh, and references the old state by the backquote syntax of vv when necessary. For our implementation, there is no need to copy the whole mesh: topological changes are deferred by means of the vertex-vertex modification queue, changes to attribute values (here, the

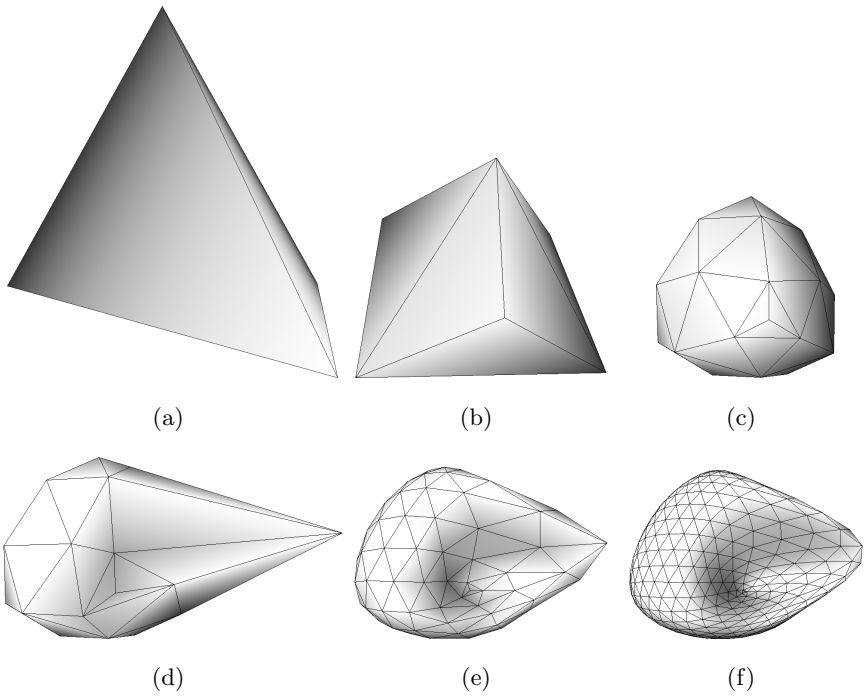


Figure 10.27. Loop subdivision: (**a**) initial tetrahedron; (**b**) result of first step; (**c**) result of second step; (**d**) interactive manipulation of vertex positions; (**e**) subdivision of manipulated mesh; (**f**) after a further step the result is quite smooth

assignment of a new position by the operator `:=` in line 15) are deferred by means of the property modification queue. As a counterpart to **synchronize**, we have to invoke `derive` in line 18 to mark the end of the current parallel derivation, i. e., to apply all collected entries of the modification queues. From a theoretical point of view, this true parallel mode is more elegant and fits well to specifications of subdivision schemes which are typically meant to be executed in parallel. From a practical point of view, it can easily be used on true parallel processors. If the number of operations is small compared to the size of the mesh, it is more memory-efficient to store the modification operations in a queue than to copy the whole mesh. Furthermore, our approach introduces a declarative, rule-based way of the specification of algorithms for polygon meshes. This can best be seen in purely topological rules like the second one in lines 20, 21, while the rule-based character is obscured by the amount of imperative code in rules like the first one which also compute and modify attribute values.

Figure 10.27 shows the application of the Loop subdivision algorithm to an initial mesh (see also Fig. 4.7(a) on page 87). It also shows a modification of vertex positions made interactively within GroIMP, the result of which is smoothed by two additional subdivision steps. It was also possible to translate the vv model of a growing apical meristem presented in [177] to XL. The outcome of the original model is shown in Fig. 4.7(b) on page 87, the result of the XL model can be seen in Fig. 10.28 on the facing page. This model is a biological model, but it differs from all other presented examples for virtual plants in that we have a detailed model of the growing surface of the apical meristem, i. e., a locally two-dimensional structure. The other models represent the plant as a locally one-dimensional structure, and its extension to three dimensions is only a matter of geometric interpretation based on attributes like length and radius.

The relative easiness of the implementation of a parallel vertex-vertex algebra together with a convenient syntax is evidence of the versatility of the XL programming language. This special application was not considered when the XL programming language was designed, nevertheless the implementation was straightforward. Contrary, the implementation of the original vv system has several limitations due to parsing problems when translating from vv source code to C++ source code [177]. The translation is done at a relatively low syntactic level where situations like the mutual nesting of vv and C++ code cannot be handled correctly. This is a further justification of the effort of defining and implementing the XL programming language as a true extension of the Java programming language, including a complete compiler.

10.7 Architecture

The XL programming language, when operating on a scene graph like that of GroIMP, can be utilized as a versatile tool to create geometry out of an

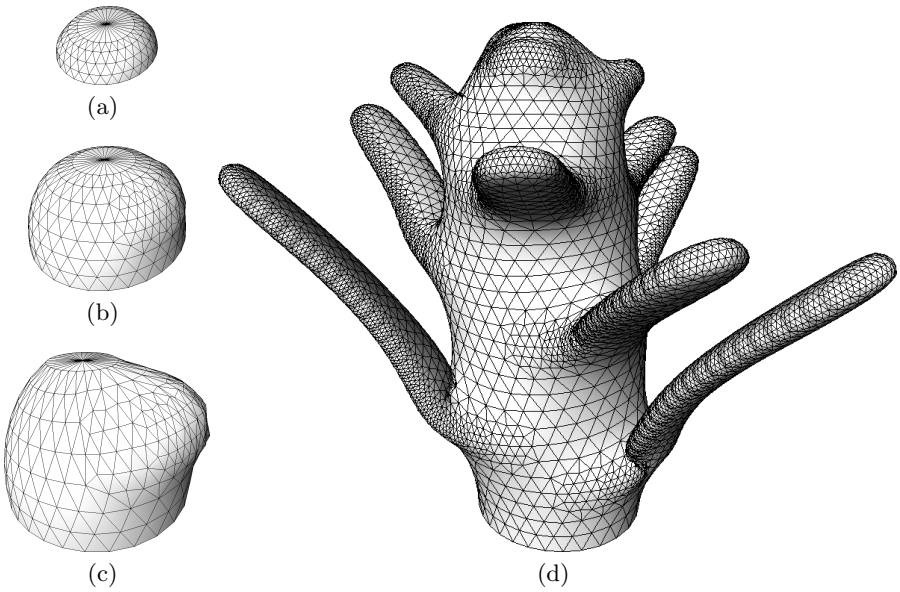


Figure 10.28. Growth of apical mersitem: (**a**) initial situation of apical meristem without primordia; (**b**) first primordium appears, subdivision of its region; (**c**) primordium grows by further subdivision at its tip; (**d**) several primordia have been created and have grown

algorithmic description. This can of course not only be used for virtual plants, but for any kind of application whose objects are defined by geometry. A prominent example is architecture, and the application of rule-based systems for architecture dates back to 1971 where *shape grammars* were described [181], originally for painting and sculpture. This was only three years after the paper of Lindenmayer in the domain of biology. In this section, we show some architectural and urban planning examples created with GroIMP. The focus lies in the aesthetic design, so it is not the intent to specify or consider the functional structure of buildings (e. g., which rooms should be connected). This can of course also be described by graphs [190, 189, 75].

10.7.1 Results of Students of Architecture

In a seminar "Artificial Growth Processes" for students of architecture at BTU Cottbus in the winter term 2006/2007, the potential of the XL programming language and GroIMP to algorithmically design (proto-)architecture was explored. The motivating idea was that the creativity of designers can be enhanced by not designing buildings themselves, but designing algorithmic processes that then create buildings. Two expectations are that the use of algorithms in design can lead to designs that would not have been conceived

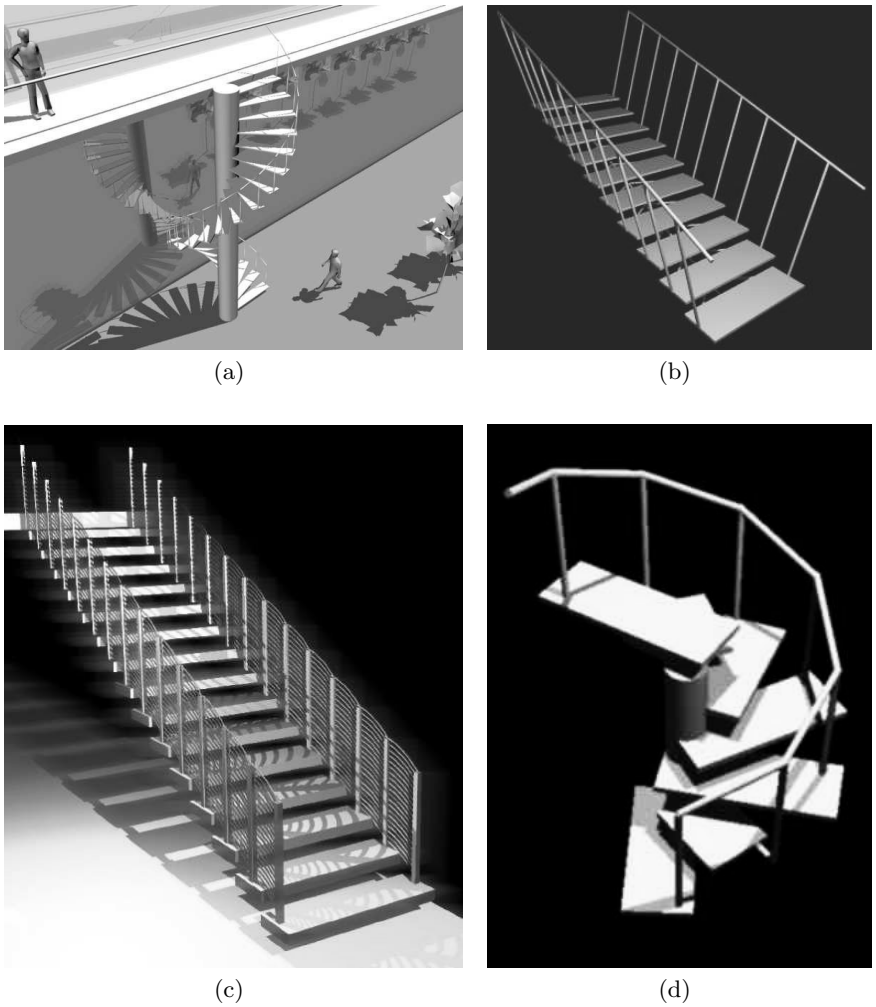


Figure 10.29. Algorithmic specification of staircases: (**a**) spiral staircase by Liang Liang, postprocessed with 3D modeller; (**b**) staircase by Simon Winterhalder; (**c**) staircase by Christopher Jarchow; (**d**) spiral staircase by Jennifer Koch

of without algorithms, especially in terms of the complexity and the combination of components, and that it is easier to adjust an algorithmic design to the specifics of a given situation [6, 97].

The latter issue was addressed by the task to create an algorithmic description of a staircase. The dimensions of the staircase were easily modifiable parameters of the XL-coded specification. Some examples are shown in Fig. 10.29.

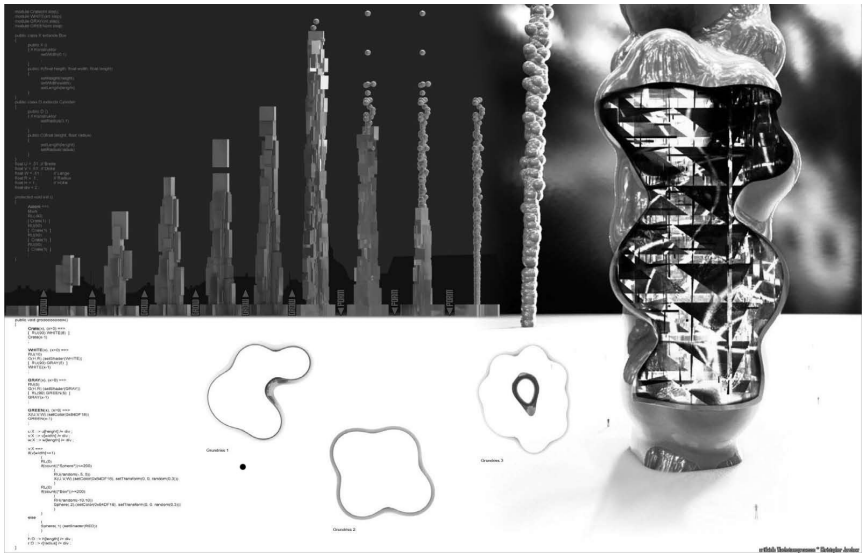


Figure 10.30. Algorithmic design study of high-rise building (by Christopher Jarchow, from [6])

The first expectation was confirmed by the observation that the development of algorithmic designs by the students "oscillated between precise intentionality and creative interpretation of unforeseen results" [6]. Figure 10.30 shows a study of an algorithmic design of a high-rise building, where a beam of boxes and spheres in vertical direction was created with GroIMP and postprocessed with another 3D modeller. Further examples can be seen in Fig. 10.31 on the following page. It has to be emphasized that for most of the students this was their first experience of programming; nevertheless, interesting results were obtained.

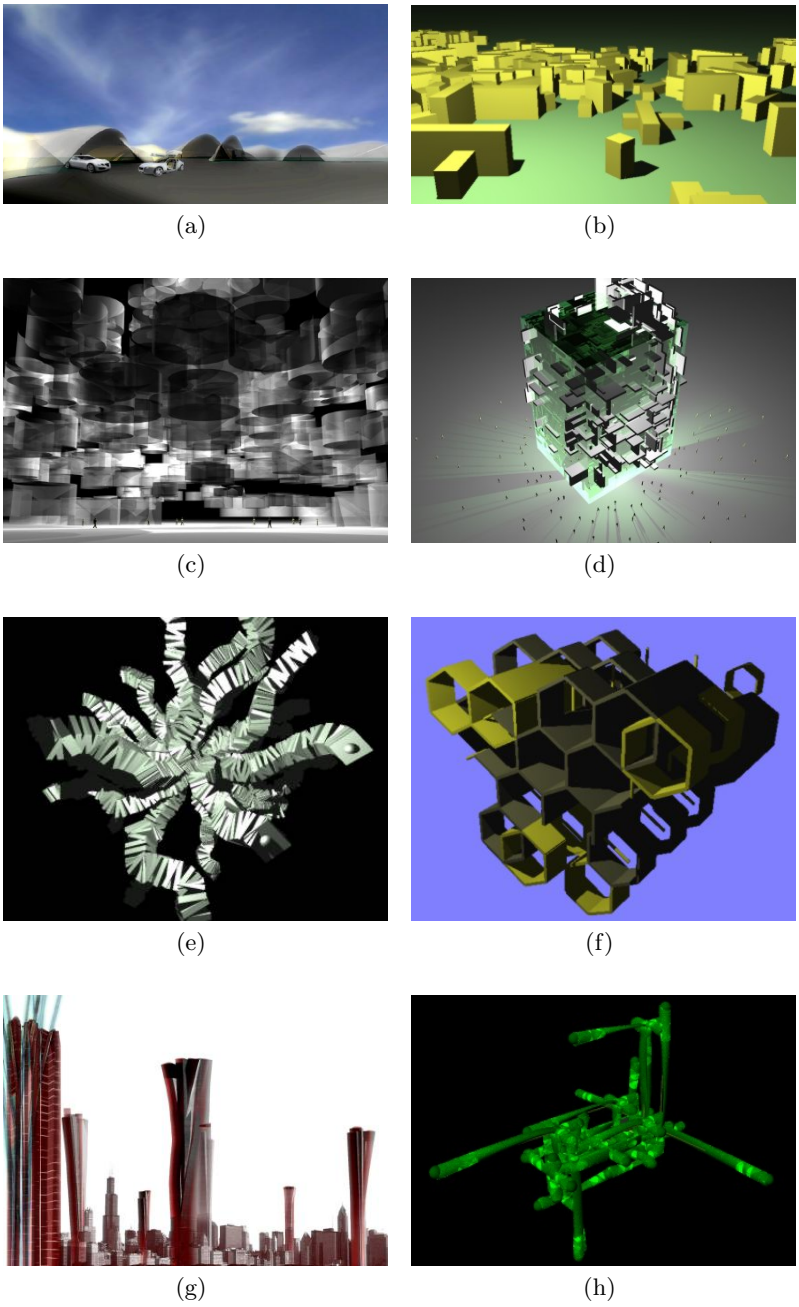
10.7.2 City Generator

The city generator was designed by the author as an example for the inclusion of context. In this toy model of urban planning, an initial context is given by existing buildings and trees. New buildings are placed on not yet occupied locations, and their outline grows until it comes close to other buildings or trees.

Context to be respected by the outline growth is represented by nodes of type `Obstacle`.

```
module Obstacle(NURBSSurface owner, float dist) extends Vertex;
```

The attribute `owner` specifies the building if the obstacle belongs to a growing building, otherwise it is **null**. The attribute `dist` defines the distance which has to be respected by growing buildings. There are three subclasses of `Obstacle`:



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 10.31. Some further results of the seminar "Artificial Growth Processes": (**a**, **g**) by Christopher Jarchow; (**b**, **e**) by Manuela Fritzsche; (**c**, **d**) by Liang Liang; (**f**) by Jennifer Koch; (**h**) by Simon Winterhalder; (**a**, **c**, **d**, **g**) were postprocessed with 3D modellers

```
module ContextPoint(float height, super.dist)
    extends Obstacle(null, dist);
module TreePoint extends Obstacle(null, 8)
    ==> Cylinder(7, 0.5).(setShader(EGA_6)) Sphere(3).(setShader(GREEN));
module OutlinePoint(super.owner, boolean done)
    extends Obstacle(owner, 15) {
    {setTransforming(false);}
}
```

`ContextPoint` nodes are placed on a regular grid at positions where the initial context already contains buildings. Their attribute `height` specifies the height of the tallest such building at the position (there may be overlapping initial buildings). `TreePoint` nodes define the location of trees and are represented by a simple cylindrical stem and a spherical crown. `OutlinePoint` nodes define the outline of growing buildings, their flag `done` indicates if growth of the point has terminated due to an obstacle in proximity. Initial context buildings have a simple box-shaped geometry and are represented by nodes of `Context`:

```
module Context(super.length, super.width, super.height, float dist)
    extends Box;
```

Now the initialization of the model creates trees along a fixed curve and places `Context` buildings at random positions without testing for overlapping with other buildings or trees. Then a regular grid is scanned for context buildings, and `ContextPoint` nodes are placed on grid positions with buildings, their `height`- and `dist`-attributes being given by the maximum of the corresponding values of the found buildings:

```
Point3d point = new Point3d(0, 0, 0.0001);
for (point.x = -200; point.x <= 200; point.x += 4)
for (point.y = -200; point.y <= 200; point.y += 4) {
    Context[] a = array((* c:Context, (point in volume(c)) *));
    float h = max(a[:][length]);
    if (h > 0) [
        ==>> ^ ContextPoint(h, max(a[:][dist])).(setTransform(point));
    ]
}
```

This regular grid is a discretization of the actual geometry of context. Ideally, GroIMP would provide algorithms to compute the (minimal) distance of a point to an arbitrarily shaped geometry, but this is not yet the case. For this reasons, we have to use such a discretization.

As a next step, the growing buildings are "sown". Within a loop, we choose a random point and test for obstacles. If the random point respects all distances to obstacles, this is the location of the next building to create, and we compute its height `h` such that heights of neighbouring context buildings are respected. Then a new `NURBSSurface` node is created to represent the new building by a generalized cylinder, its profile being given by a sequence of initially ten `OutlinePoint` nodes arranged on a circle.

```
point = new Point3d(random(-150, 150), random(-150, 150), 0.0001);
if (empty((* o:Obstacle(, d), (point.distance(o) < d) *))) {
    double h = lognormal(0.5, 0.1)
        * Math.min(min(((* c:ContextPoint *),
                        c[height] * point.distance(c) / c[dist])),
                   100);
    [
        ==>> ^ s:NURBSSurface(...)
            [   for (int i : 0 : 9) (
                    OutlinePoint(s, false)
                        .(setTransform(5 * Math.cos(i*Math.PI/5),
                                       5 * Math.sin(i*Math.PI/5), 0))
                )
            ];
    ]
}
```

A possible result after initialization is shown in Fig. 10.32.

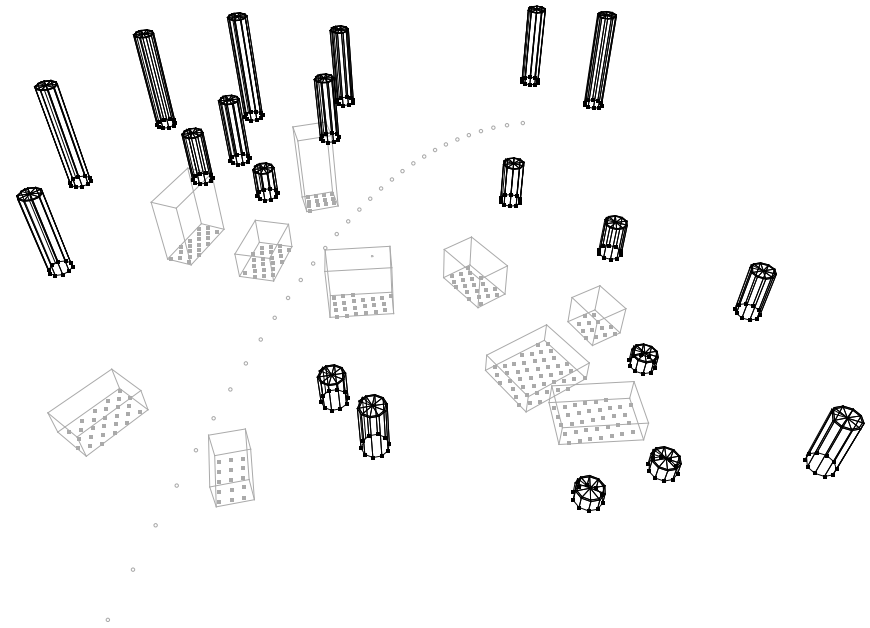


Figure 10.32. City generator after initialization. Trees are shown as circles, context buildings as grey boxes with context points at their floor. The new buildings (black cylinders) are placed such that they respect distances to the complete context. Furthermore note that they also respect the height of the initial context buildings: close to these buildings, the height of the cylinders is small, while it increases towards the back. During growth, the black points defining the outline of the (generalized) cylinders will be moving away from the centre until they come into proximity of an obstacle

The growth of buildings is specified by two rules. The first one centrifugally shifts outline points away from the centre as long as there is no close obstacle and a maximum size of 20 has not yet been reached. If this termination condition is fulfilled, the flag `done` is set to prevent further handling of the outline point.

```
v:OutlinePoint(owner, false) ::>
    if ((v[position].length() < 20)
        && empty((* o:Obstacle(oo, d),
                    ((owner != oo) && (distance(v, o) < d)) *))) {
        v[x] :*= 1.1;
        v[y] :*= 1.1;
    } else {
        v[done] := true;
    }
```

The second growth rule refines the list of outline points by the insertion of new outline points where the distance of neighbouring outline points has exceeded a threshold of 10.

```
v:OutlinePoint(owner, vdone) w:OutlinePoint(, wdone),
(!(vdone && wdone) && (distance(v, w) > 10)) ==>>
    v
    OutlinePoint(owner, false)
        .(setTransform((v[position]+w[position])*0.51))
    w;
```

Figure 10.33 on the following page shows a result of the model from different perspectives and with different lighting.

10.8 AGTIVE '07 Tool Contest

This section presents two applications which were developed for the tool contest of the AGTIVE '07 conference [164], and it shows a benchmark of the solutions of the third task of the contest, Sierpinski triangles, whose XL-based implementations were already described in Sect. 9.2.1 on page 253 and Sect. 10.1.2 on page 272. Although the domains of the two applications, game specification and UML-related model transformation, are quite different from the primary domain of plant model specification, the XL programming language and GroIMP allowed to concisely solve the tasks of the contest. This is further evidence that our system is useful for a quite general range of applications, given that we can take advantage of the graph structure and transformation rules.

10.8.1 Ludo Game

The goal of this case study was to model the German variant "Mensch ärgere dich nicht" of the Ludo game [163]. We do not give a description of the

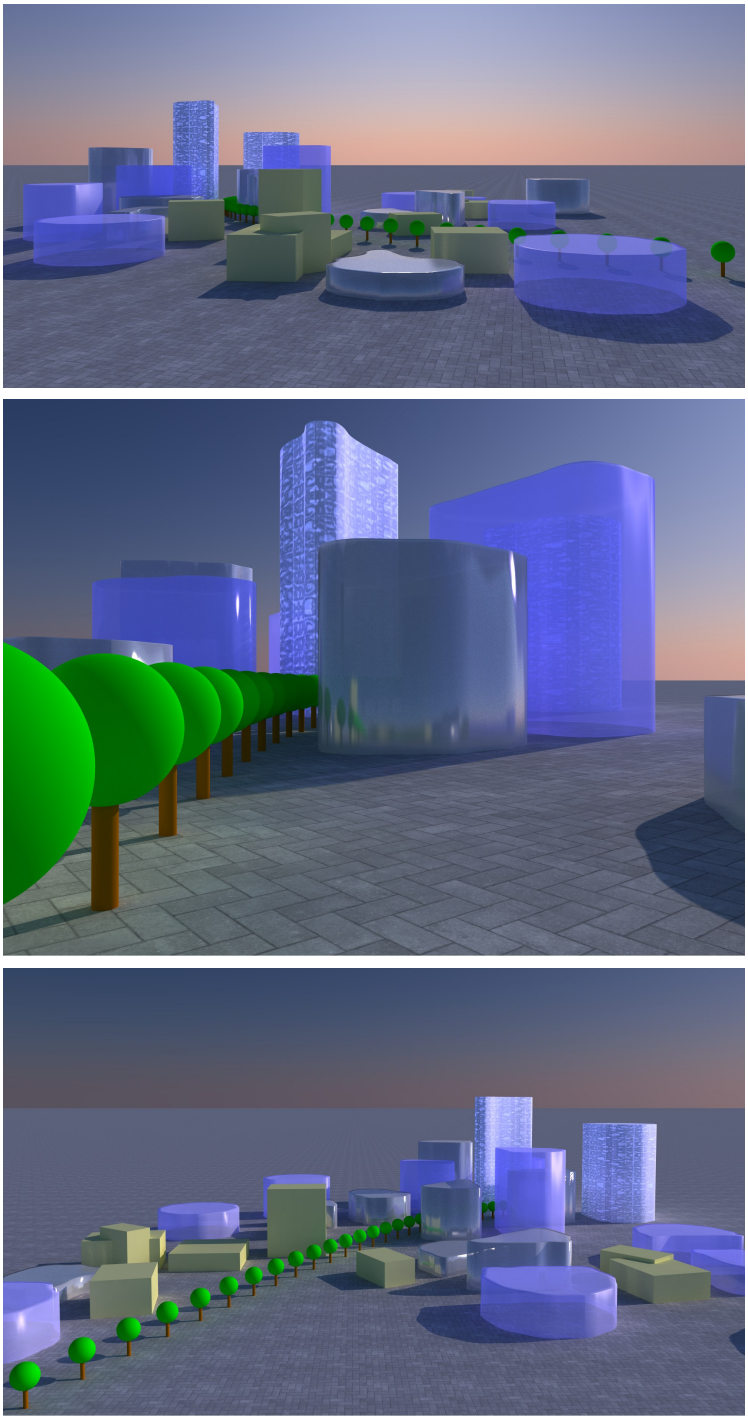


Figure 10.33. Result of city generator. Note how buildings grew into gaps

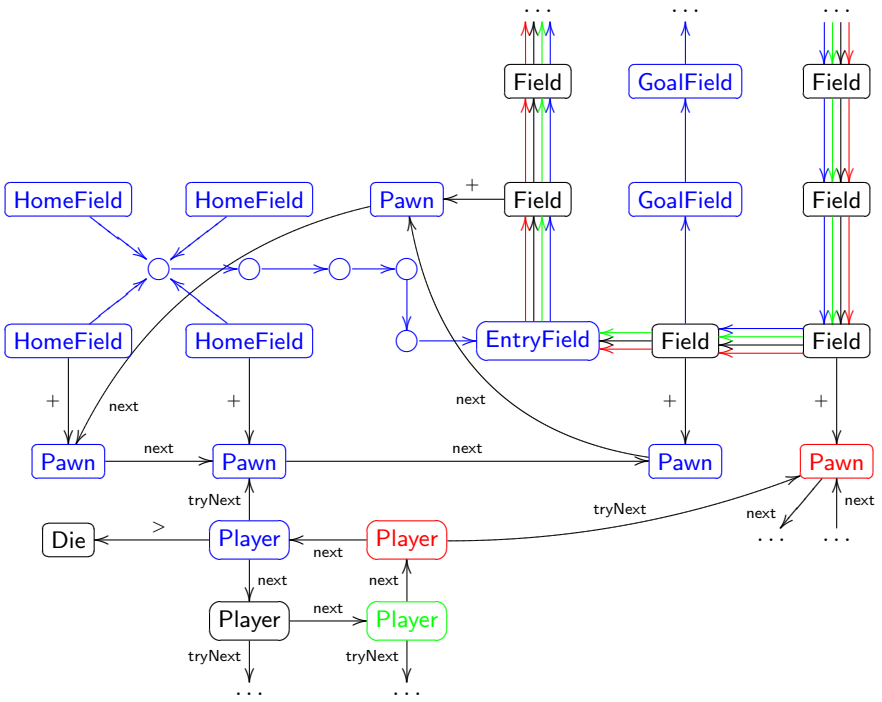


Figure 10.34. Part of graph of the Ludo game. Yellow shown as black

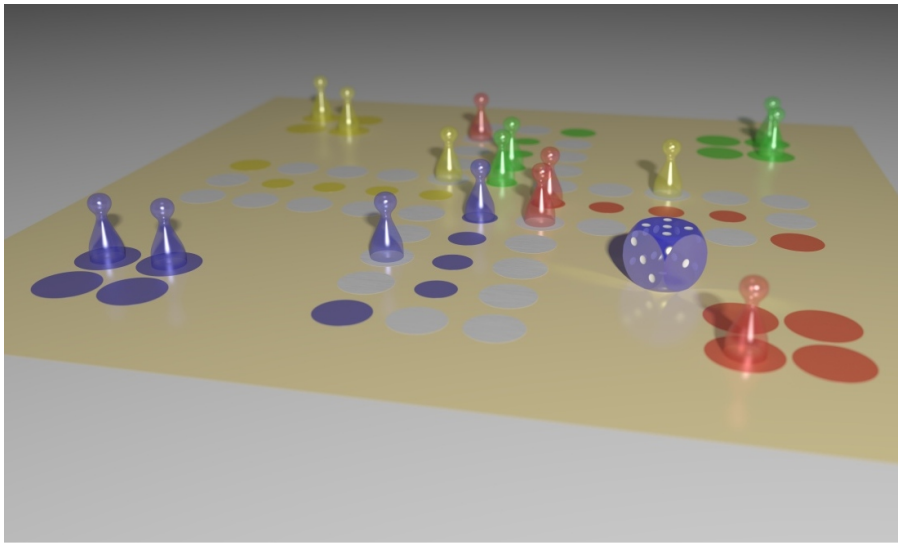


Figure 10.35. Rendered view of Ludo board

specification of the case study as it can be found in the cited literature, but show the main rule of our implementation and explain which rules of the game are expressed by this single rule.

But at first, let us consider the structure. Figure 10.34 on the previous page shows a part of the structure, the meaning of the components is described in the following. We represent the four players by nodes of class `Player` and assign indices from 0 to 3 to them. Players are organized in a cyclic list via edges of type `next`.

```
const int next = EDGE_4;
module Player(int index);
```

There is a single `Die` node with a linear congruential pseudorandom number generator as specified by the case study. A die is also drawn in 3D, but we omit the discussion of geometry-related aspects.

```
module Die extends Null {
    int rand;

    int score() {
        return (rand >> 8) % 6 + 1;
    }

    void nextScore() {
        do {
            this[rand] = 1664525 * rand + 1013904223;
        } while ((rand < 0) || ((rand >> 8) >= 0x7ffffe));
        ... // computation of rotation matrix
    }
} ==> ... ; // representation by geometry
```

At every point in time, it is the turn of one player. This is indicated by a successor edge from the player to the single die.

The board is represented by `Field` nodes. These are connected by edges of the types `redEdge`, `blueEdge`, `yellowEdge`, `greenEdge`.

```
const int redEdge = EDGE_0;
const int blueEdge = EDGE_1;
const int yellowEdge = EDGE_2;
const int greenEdge = EDGE_3;
const int[] edges = {redEdge, blueEdge, yellowEdge, greenEdge};

module Field(super.shader)
    extends Cylinder(0.001, 0.4).(setShader(shader)) {
    { ... } // initialize transformation properties
}
```

An edge for player p exists from field a to field b if and only if p may move its own pawns from a to b. I. e., normally edges of all four types are present between neighbouring fields, but the field immediately in front of the entry field of a player acts as a junction where this player may only move pawns to

the goal fields, whereas other players have to move pawns to the neighbouring entry field. Also the home fields, where pawns are initially placed, are represented as nodes. From these there is a path of six edges to the entry field (see Fig. 10.34 on page 349). Entry, goal and home fields are represented by special classes:

```
abstract module PlayerField(Player player)
    extends Field(colors[player.index]);
module EntryField(super.player) extends PlayerField;
module GoalField(super.player) extends PlayerField;
module HomeField(super.player) extends PlayerField;
```

Pawns are nodes of class `Pawn`. Each pawn is located at a single field, indicated by a branch edge from the field to the pawn. Like players, the pawns of each player are organized in a circular list of `next` edges. There is an edge of type `tryNext` from each player to one of his pawns. This edge specifies the pawn which the player should try to move at first when it is his turn. The case study defines also variants of the game with more sophisticated strategies, but we do not describe their implementation here. Nevertheless, they are part of the Ludo game in the example gallery.

```
const int tryNext = EDGE_5;
module Pawn(Player player, HomeField home)
    extends NURBSSurface(surface("Pawn"))
    .(setShader(colors[player.index]));
```

The initialization of the model creates the complete board including pawns, players and the die according to Fig. 10.34 on page 349. The main rule of the game is responsible for the movement of a pawn and looks as follows:

```
(* d:Die < p:Player *) -tryNext-> (* t:Pawn *) (-next->){0,3} : (
    (* f:Pawn [-next-> n:Pawn] *)
    <+ (* a:Field *) (-edges[p.index]->){d.score()} b:Field
    (? +> g:Pawn(gp,h)), ((g == null) || (gp != p))
), n|t
    ==>>
    b [f], p -tryNext-> n, if(g != null) (h [g]);
```

Its left-hand side is quite complex. In order to understand it, we can ignore the starred parentheses `(* *)` for the moment as they only indicate which part of the left-hand side shall be in the context, but have no influence on the structure of the left-hand side (Sect. 6.5.8 on page 154). The pattern starts with the die to which a successor edge shall point from a player. The matching player `p` is thus the current player which has to move a figure. Via a `tryNext` edge we find the pawn `t` which shall be tried at first for a legal move. However, this pawn is not necessarily the one which is actually moved: for the latter we may have to traverse up to three `next` edges in the circular list of pawns until we find a pawn which can move legally. This search for a movable pawn is implemented by `(-next->){0,3} : (...)` (see Sect. 6.5.6 on page 152): this pattern traverses up to three `next` edges and stops when it finds a match for

the subpattern in parentheses, which specifies a legal move. For a legal move, the pawn `f` to be moved has to have a next pawn `n` in the circular list, which is always the case, and it has to be located on a field `a` from which we can reach a field `b` by traversing exactly `d.score()` edges of the corresponding player edge type. Then an optional (indicated by `?`) pattern follows in parentheses: there may be a pawn `g` on `b`. Finally, we have the condition that if such a pawn `g` has been found, it must not belong to the same player (namely, because such a pawn is kicked and returns to its home field, but a player cannot kick his own pawns). We also have to specify the folding clause `n|t` (Sect. 6.5.9 on page 155) because it may happen that the pawn `n` following the moved pawn `f` coincides with the first tried pawn `t`, i. e., the match may be noninjective with respect to `n`, `t`.

Now if there is a match, its non-context parts are replaced by the right-hand side. `b [f]` creates a `branch` edge from field `b` to pawn `f` in order to move the pawn. Then we establish a `tryNext` edge from player `p` to the pawn `n` which follows `f` in the circular list, and finally we move `g` to its home field `h` if a match for `g` was found. The commas separate parts of the right-hand side so that no implicit successor edge is created between them. In total, the single rule, together with the used graph structure, implements the following rules of the Ludo game:

1. A player may move one of its pawns in the game (i. e., not on a home field) forward by the exact score.
2. If the score is six, a player may move a pawn on a home field to the entry field. This rule is implemented by the special path of six edges from home fields to the entry field. Its intermediate nodes are not of class `Field` but of class `Node` so that a forbidden stay of a pawn on such a node cannot occur.
3. The target field of a move must not be occupied by a pawn of the moving player. If it is occupied by a pawn of another player, this pawn is kicked and returns to a home field of its player.
4. After a complete round, a pawn is not moved to the entry field, but enters the goal fields. This is implemented by the player edges defining possible moves for the individual players.
5. If a player can move legally, he must do so.

The movement rule is completed by a rule which moves the die to the next player unless the score is six, and by a rule which throws the die:

```
d:Die < (* p:Player -next-> q:Player *), (d.score() != 6) ==>> q d;
d:Die ::> d.nextScore();
```

Rule application is controlled by control flow statements:

```
this[move]++;
[
    ... // rule for pawn movement
]
```

```
derive();
if(count((* Die < p:Player, GoalField(p) [Pawn] *)) == 4) {
    println("Player " + p.index + " won after " + move + " moves.");
} else [
    ... // die rules
]
```

Figure 10.35 on page 349 shows a rendered view of the Ludo board after several rounds.

Besides the described fully automatic game simulation, we also implemented a human player. The code can be seen in the example gallery of GroIMP. Here we only show a feature useful for games and other interactive applications: methods can be annotated with `@de.grogra.rgg.Accelerator` to associate a shortcut key to the invocation of the method as in

```
@Accelerator("F12")
public void run() {
    ...
}
```

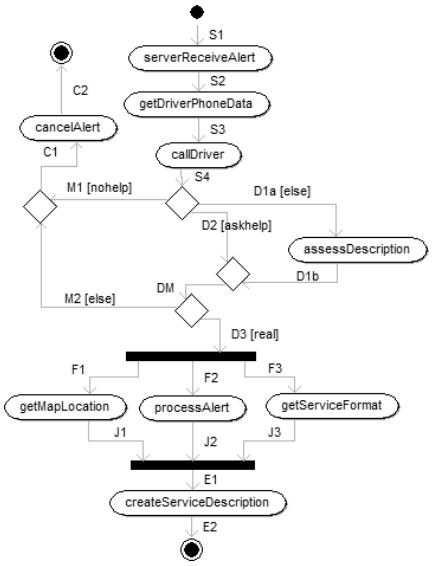
I. e., the user can fully and quickly control an application within GroIMP by pressing suitable keys.

10.8.2 Model Transformation from UML to CSP

This example implements a model transformation from UML activity diagrams [136] to communicating sequential processes (CSP) [79] as specified in [193]. A UML activity diagram typically describes the low-level behaviour of software components, for an example see Fig. 10.36(a) on the following page. The verification of such behaviour is an important issue. Communicating sequential processes are useful for this purpose [193]: an activity diagram is transformed into equivalent communicating sequential processes which can be used for verification as there is a formal semantics for CSP. Processes are given by equations, Fig. 10.36(b) shows the equivalent of the activity diagram. But equations can of course be represented as trees, so the transformation from activity diagrams to CSP can be implemented as a graph transformation.

The meta models for activity diagrams and CSP are shown in Fig. 10.37, they are explained in more detail together with all required transformations in [193]. The most important transformation is that UML activity edges like the edge S_1 from the initial node to the action named serverReceiveAlert are represented by process identifiers in CSP. Each such identifier appears on the left-hand side of a process equation. As more complex examples, consider the transformation for actions in Fig. 10.38(a) and for decision nodes in Fig. 10.38(b).

The starting point for the implementation of the transformation using the XL programming language is the definition of the meta models. As neither the XL programming language nor GroIMP currently allow to specify type

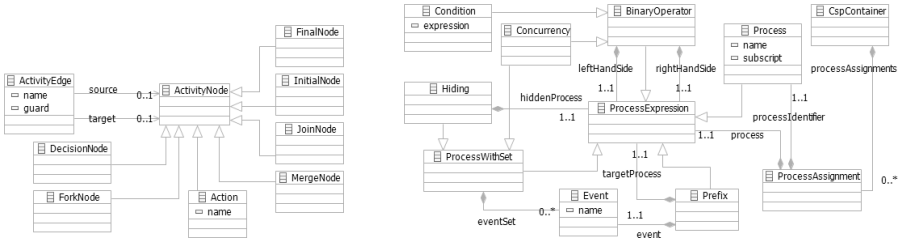


(a)

$$
\begin{aligned}
S_1 &= \mathsf{serverReceiveAlert} \rightarrow S_2 \\
S_2 &= \mathsf{getDriverPhoneData} \rightarrow S_3 \\
S_3 &= \mathsf{callDriver} \rightarrow S_4 \\
S_4 &= M_1 \not\leftarrow \mathsf{nohelp} \not\rightarrow (D_2 \not\leftarrow \mathsf{askhelp} \not\rightarrow D_{1a}) \\
M_1 &= C_1 \\
D_2 &= D_M \\
D_{1a} &= \mathsf{assessDescription} \rightarrow D_{1b} \\
D_{1b} &= D_M \\
D_M &= D_3 \not\leftarrow \mathsf{real} \not\rightarrow M_2 \\
M_2 &= C_1 \\
C_1 &= \mathsf{cancelAlert} \rightarrow C_2 \\
C_2 &= \mathsf{SKIP} \\
D_3 &= F_1 \parallel F_2 \parallel F_3 \\
F_1 &= \mathsf{getMapLocation} \rightarrow J_1 \\
F_2 &= \mathsf{processAlert} \rightarrow J_2 \\
F_3 &= \mathsf{getServiceFormat} \rightarrow J_3 \\
J_1 &= \mathsf{processJoin} \rightarrow E_1 \\
J_2 &= \mathsf{processJoin} \rightarrow \mathsf{SKIP} \\
J_3 &= \mathsf{processJoin} \rightarrow \mathsf{SKIP} \\
E_1 &= \mathsf{createServiceDescription} \rightarrow E_2 \\
E_2 &= \mathsf{SKIP}
\end{aligned}
$$

(b)

Figure 10.36. Model transformation (from [193]): (**a**) UML activity diagram; (**b**) equivalent list of communicating sequential processes



(a) (b)

Figure 10.37. Meta models (from [193]): (**a**) (subset of) UML activity diagram meta model; (**b**) (subset of) CSP meta model

graphs, i. e., to define restrictions on the allowed source and target types as well as multiplicities of edge types, the definition of a meta model amounts to the definition of suitable types for nodes and edges. By using module declarations for node types and definitions of **int**-constants for edge types, the translation to XL code is straightforward. One exception is that we represent activity edges of UML activity diagrams as nodes within the graph of GroIMP, although they stand for edges in the diagram:

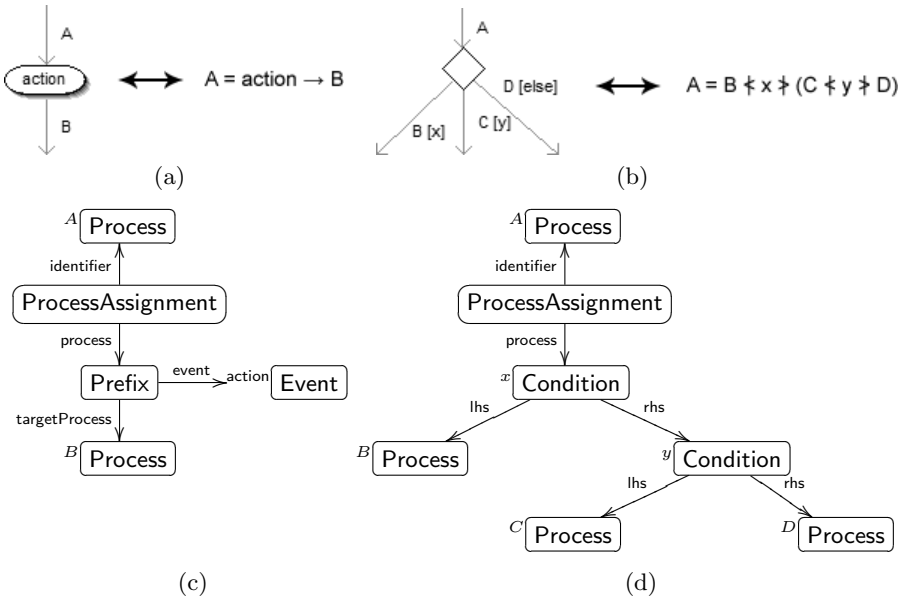


Figure 10.38. Rules for model transformation [193]: (**a**) UML actions are transformed to CSP events; (**b**) UML decision nodes are transformed to (possibly nested) CSP conditions; (**c**) tree of CSP expression of (a); (**d**) tree of CSP expression of (b)

```
module ActivityEdge(String n return getName(), String guard)
    extends Node.(setName(n)) {
    ActivityEdge(String name) {
        this(name, null);
    }
}
```

But this implementation detail is hidden by the syntax. Already for the ant simulation (Sect. 10.3.2 on page 288) we used nodes as if they were edges. In Sect. B.7 on page 401 it is discussed that such an auxiliary node together with an incoming edge of the special type `EDGENODE_IN_EDGE` and an outgoing edge of the special type `EDGENODE_OUT_EDGE` can be addressed by the syntax for edges. Here we define shortcuts for these edge types as we will frequently need them:

```
const int o = Graph.EDGENODE_OUT_EDGE;
const int i = Graph.EDGENODE_IN_EDGE;
```

Another difference to the original meta model is that we introduce an abstract node class `BranchNode` as a superclass of `ForkNode` and `DecisionNode`.

The transformation for actions (Fig. 10.38(a)) is specified by the following rule:

```
a:ActivityEdge -o-> x:Action -i-> b:ActivityEdge ==>>
    ^ -processAssignments-> ProcessAssignment [-identifier-> a]
    -process-> Prefix [-event-> Event(x.getName())] -targetProcess-> b;
```

Its right-hand side already has the complete structure of the corresponding CSP expression (see Fig. 10.38(c) on the preceding page). However, instead of CSP process identifier nodes there are still UML activity edges `a`, `b`. As these are in a one-to-one correspondence, we may keep activity edges in the graph and replace them only at the end by process identifier nodes.

The rule for branch nodes (i. e., fork and decision nodes) only creates an interim structure with the branch node at the location where an expression tree should be at the end (see Fig. 10.38(d) and Fig. 10.38(b)):

```
a:ActivityEdge -o-> b:BranchNode ==>>
    ^ -processAssignments-> ProcessAssignment [-identifier-> a]
    -process-> b;
```

In order to construct a correctly nested binary tree for the (arbitrary) number of children of branch nodes, we have to sequentially apply the two rules

```
b:DecisionNode -i-> a:ActivityEdge, (!"else".equals(a.guard)) ==>>
    c:Condition(a.guard) [-lhs-> a] -rhs-> b
    moveIncoming(b, c, -1);

b:ForkNode (* [-i-> ActivityEdge] *) -i-> a:ActivityEdge ==>>
    c:Concurrency [-lhs-> a] -rhs-> b
    moveIncoming(b, c, -1);
```

Both rules pick an arbitrary child `a` of the branch node `b` with the condition that this must not be the else-branch in case of a decision node or that there must be another child (placed in context parentheses) in case of a fork node. Note that although there is a non-determinism regarding the order of picked children, all possible CSP expressions are semantically equivalent as guard conditions of decision nodes shall be disjoint [193]. Then a condition or concurrency node, respectively, is created with the chosen `a` as left-hand side and the still intermediate `b` as right-hand side, and it replaces `b` by invoking `moveIncoming`, i. e., by moving all incoming edges from `b` to `c` on derivation (see also the explanation on page 314). This sequential replacement is done as long as possible, i. e., until all branch nodes have only one child. Then branch nodes can be removed:

```
BranchNode -i-> a:ActivityEdge ==> a;
```

The procedure is illustrated in Fig. 10.39 on the next page. In Fig. 10.39(f) we also see the final step where activity edges are replaced by process identifiers:

```
a:ActivityEdge ==> Process(a.getName());
```

The whole transformation is specified as a sequence of blocks of the previously described rules. Each block leads to parallel or sequential production, depending on the derivation mode (see Sect. 9.1.5 on page 245).

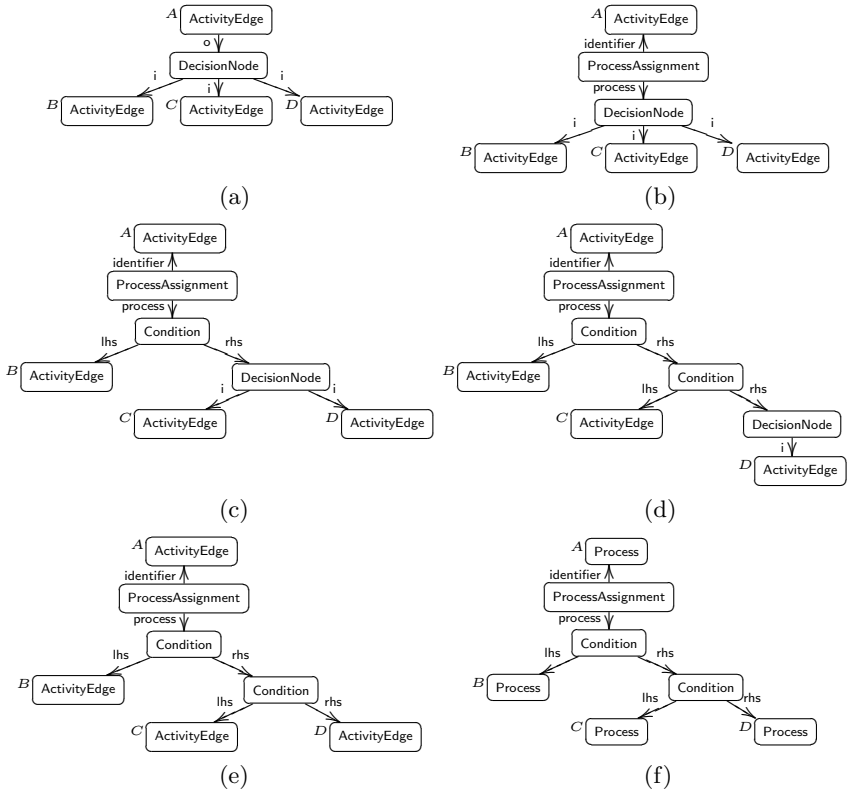


Figure 10.39. Transformation from UML decision node to CSP expression: (**a**) initial UML structure; (**b**) first rule created process assignment with decision node as interim child; (**c**), (**d**) sequential creation of binary tree; (**e**) deletion of decision node; (**f**) final replacement of activity edges by process identifiers

```
setDerivationMode(PARALLEL_MODE);
[
    ... // main transformation rules
]
derive();
setDerivationMode(SEQUENTIAL_MODE);
for (applyUntilFinished()) [
    ... // sequential creation of binary tree for decision/fork nodes
]
derive();
setDerivationMode(PARALLEL_MODE);
[BranchNode -i-> a:ActivityEdge ==> a;]
derive();
[a:ActivityEdge ==> Process(a.getName());]
derive();
```

For a concrete model transformation, we need an activity diagram as input. This can be either specified as fixed part of the source code, which is not very useful but simple and concise due to the convenient syntax. For the example of Fig. 10.36(a) on page 354, we initialize the graph with

```
Axiom ==>>
    ^ InitialNode
    -ActivityEdge("S1")-> Action("serverReceiveAlert")
    -ActivityEdge("S2")-> Action("getDriverPhoneData")
    ... ;
```

Alternatively and more useful from a practical point of view is the input of the activity diagram by a file in a common data format for graphs. The GraphML-import of GroIMP (Sect. A.6.1 on page 388) can be used for this purpose. For the example diagram, we specify the graph like

```
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
  <key id="ntype" attr.name="type" for="node"/>
  <key id="nval" attr.name="value" for="node"/>
  <key id="etype" attr.name="type" for="edge"/>
  <key id="eval" attr.name="value" for="edge"/>
  <node id="i">
    <data key="ntype">UMLToCSP$InitialNode</data>
  </node>
  <edge source="i" target="sra">
    <data key="etype">UMLToCSP$ActivityEdge</data>
    <data key="eval">name="S1"</data>
  </edge>
  <node id="sra">
    <data key="ntype">UMLToCSP$Action</data>
    <data key="nval">name="serverReceiveAlert"</data>
  </node>
  <edge source="sra" target="gdpd">
    <data key="etype">UMLToCSP$ActivityEdge</data>
    <data key="eval">name="S2"</data>
  </edge>
  <node id="gdpd">
    <data key="ntype">UMLToCSP$Action</data>
    <data key="nval">name="getDriverPhoneData"</data>
  </node>
  ...
</graphml>
```

To obtain a textual output of all CSP process equations, we equip CSP classes with `expr` methods that return a `String` representation of the expression. As an example, a `BinaryOperator` searches for its two children and combines their textual representation with the operator symbol:

```
abstract module BinaryOperator extends ProcessExpression {
    abstract String op();
```

```
    String expr() {
        for((* this [-lhs-> l:ProcessExpression]
                    -rhs-> r:ProcessExpression *))
        {
            return '(' + l.expr() + op() + r.expr() + ')';
        }
        return toString(); // lhs, rhs not found (incomplete structure)
    }
}
```

This example for a model transformation differs from all other presented examples. We do not model the growth of a structure out of a seed or the dynamics of a structure, we rather take a given graph as input and translate this input into another graph as output. Thus, this example is not related to the principal application domain of the XL programming language. Nevertheless, it is easily possible to implement the transformation.

10.8.3 Sierpinski Triangles Benchmark

We already presented two implementations of the relational growth grammar of Ex. 5.23 on page 113: the implementation in Sect. 9.2.1 on page 253 on the basis of a lightweight custom graph and the implementation in Sect. 10.1.2 on page 272 on the basis of the graph of GroIMP. Both were developed for the Sierpinski triangle case study of the AGTIVE '07 tool contest. The aim of this case study was to provide a simple benchmark for graph transformation tools with respect to both space and time efficiency [191].

The benchmark allows us to compare both the different XL-based solutions, but also the XL-based solutions with other graph transformation tools. To perform the required measurements, we extended the examples by automatic invocations of the method `System.nanoTime()` which returns the value of the system timer in nanoseconds, and added textual output of the elapsed time per step. Furthermore, we prepended a warm-up round of ten steps so that the JIT compiler of the Java virtual machine is triggered for the most frequently used methods. For the GroIMP-based solution, we used the pure topological variant without 3D geometry as the latter introduces both a space and time overhead. Furthermore, we created a variant where the change log of GroIMP, which is normally active and supports graph change listeners and an undo function, is deactivated in order to reduce the space and time overhead.

Each benchmark variant was executed ten times on a 3 GHz Intel Xeon 5160 with 16 GiB RAM, JRE 1.6.0_06, SUSE Linux 10.0 and GroIMP 0.9.8.1 by the command line `java -server -Xms2048m -Xmx2048m -jar core.jar`. Note that although the computer has 16 GiB RAM installed, the 32-bit Java virtual machine can only use about 2.6 GiB RAM, and we set the maximum size of the heap to 2 GiB by the shown command line options. The average of the measured running times per step is shown in Table 10.2 and Fig. 10.40.

Table 10.2. Running times per step in milliseconds (average of ten runs). Missing data indicates an `OutOfMemoryError`

step	GroIMP graph	GroIMP graph without log	custom graph	**int** graph
1	0.49	0.39	0.17	1.23
2	0.43	0.34	0.30	0.15
3	0.55	0.41	0.33	0.19
4	0.92	0.64	0.50	0.37
5	2.05	1.26	0.96	1.04
6	5.55	3.21	2.54	2.69
7	15.93	9.01	8.12	9.96
8	45.30	26.25	22.80	24.70
9	141.46	76.99	56.41	51.53
10	561.13	255.31	91.77	95.39
11	1,471.80	830.22	284.33	286.01
12	5,151.97	2,770.05	879.73	851.56
13	16,491.18	8,697.52	2,711.16	2,583.74
14		29,184.79	8,761.59	7,412.75
15			26,271.45	22,208.07
16				66,328.92
17				199,097.41

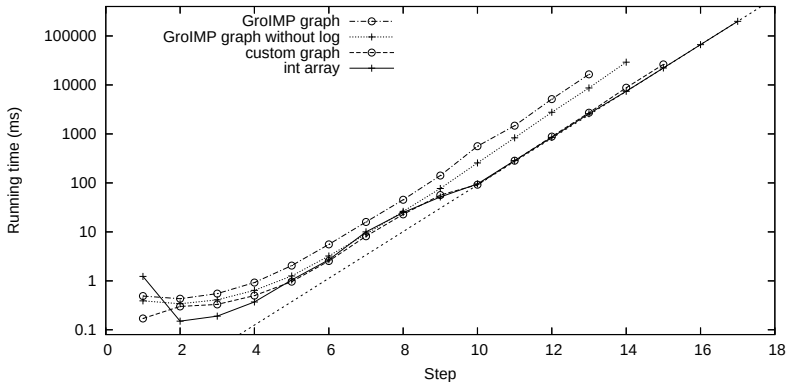


Figure 10.40. Plot of Table 10.2 with logarithmic y-axis. Dashed line depicts an exact exponential law with base 3

The number of executed steps is limited by the available memory, i. e., the next step would result in an `OutOfMemoryError`. Note that the plot uses a logarithmic y-axis, which results in a roughly linear graph. This is as expected as the number of nodes after n steps is $\frac{3}{2}(3^n + 1)$ and the number of edges 3^{n+1}, i. e., we have an exponential growth. Furthermore, the comparison with

the straight dashed line shows that the factor for the increase of running times is roughly 3, which is also as expected.

For all variants, the limiting factor in the end is the available memory. Therefore, an even more lightweight representation of the graph than that of the example of Sect. 9.2.1 is desirable. On a 32-bit Java virtual machine, an instance of its class `Vertex` requires 12 bytes for the three pointers, 4 bytes for the `index`-variable of the superclass `Node` and 8 bytes for internal information of the Java virtual machine, i. e., 24 bytes in total. A closer look at the grammar shows that each vertex has at most two outgoing edges, so that two pointers suffice given that we know the type of their edge. In languages like C++ we could store the type information in unused bits of the pointers, and we could also get rid of the 8 bytes for internal information if we used a final class. The Java programming language does not allow such low-level tricks, but we can simulate a low-level access to memory by storing the graph in a large **int**-array where two consecutive values represent the two pointers of a vertex, together with their type information. A vertex is then addressed by its index into this array. In our implementation of this trick, we use the two least significant bits of an **int**-value to encode the type of the edge (none, $\mathtt{e}_0$, $\mathtt{e}_{120}$, $\mathtt{e}_{240}$), while the remaining 30 bits encode the index of the target vertex. This allows up to 2^{30} vertices or 18 steps, and each vertex consumes 8 bytes on a 32-bit machine.

The remaining problem is how to implement the XL interfaces for such a graph representation. This can be done in a relatively straightforward way by using the simple implementation again and proxy objects for vertices. I. e., whenever there is need to pass a vertex to the XL run-time system, the original representation as an **int**-valued index into the large array is wrapped into a `Node` object whose methods operate on the array. But then the problem is that due to the parallel derivation mode all modifications of the current step are collected in queues using these proxy objects, and that the number of modifications is of the same magnitude as the size of the graph. Thus, the modification queues and proxy objects occupy more memory than the graph itself. This problem would not occur if we had a sequential derivation mode, and in fact for this specific grammar we obtain the same result if we immediately apply changes for each match of the original graph. Our solution is a mixture: it collects changes of a relatively small number of 1,000 matches and applies them to the graph in one go by invoking the `derive` method within the execution of a right-hand side. The result of this variant is also shown in the table and figure. We can achieve two further steps within 2 GiB RAM due to the compact graph representation and the negligible size of the queue collection. For the reached 17 steps, the required size of the **int**-array on a 32-bit machine is $8 \cdot \frac{3}{2}\left(3^{17}+1\right)$ bytes ≈ 1.44 GiB.

More interesting than the comparison of different XL-based implementations is the comparison of these implementations with solutions based on other graph transformation tools. Figure 10.41 on the following page shows the result of the case study [191], where we have replaced our own measure-

ments by the newer and faster ones of Table 10.2. The LIMIT solution is a hand-coded implementation in the C programming language and uses a graph representation similar to our **int**-array. This solution can be regarded to be (nearly) the fastest possible. Like the second fastest solution, the Fujaba implementation, matches are not found automatically, but have to be provided by the developer. The third fastest solutions in the figure are our custom graph and **int**-array solutions, which find their matches automatically. The same holds for the following group of GrGen.NET, Two Tapes and the two GroIMP-based solutions of comparable running times. Note that an objective comparison is not possible as the measurements shown in the figure were carried out on different machines. It might even be the case that if the GrGen.NET or Two Tapes solutions would be executed on the same machine as our fastest solutions, we would obtain comparable running times. Although this seems to be unlikely given the factor of about 6, we cannot conclude that our implementations are the fastest ones except Fujaba and LIMIT, but we can conclude that they belong to the group of fastest ones with automatic pattern matching. This group is several orders of magnitude faster than the least efficient solutions, whose time complexity is not even of the form $\mathcal{O}(3^n)$.

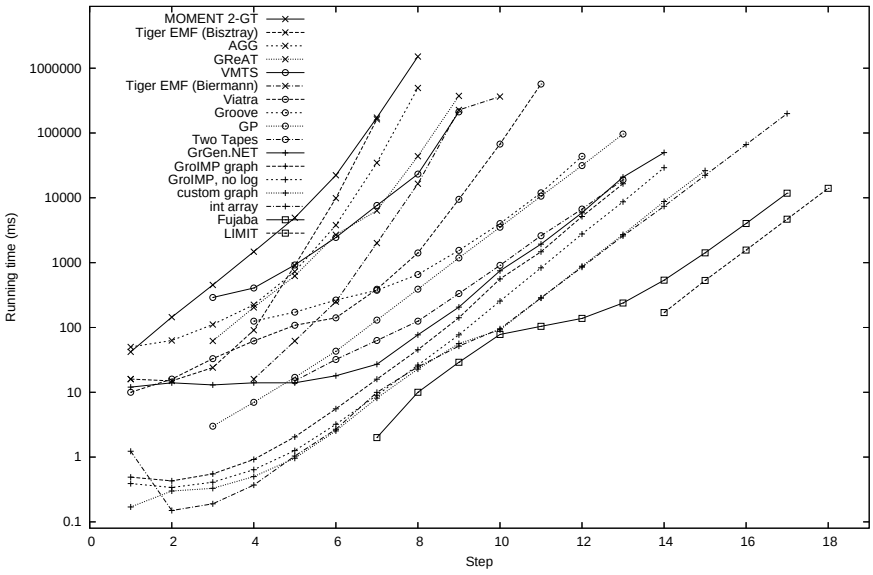


Figure 10.41. Running times of solutions of the Sierpinski case study (carried out on different machines). Data from Table 10.2 and [191]

11

Discussion

In this chapter, we give some concluding remarks on the present work, and indicate possible future extensions at the end. This discussion shall clarify the achievement of this work: how it builds on the well-established parts L-systems, graph grammars and the Java programming language to define a variant of graph grammars and a new programming language, and why this is a suitable basis for current and future functional-structure plant modelling.

11.1 Relational Growth Grammars

The main result of the first part of this work, relational growth grammars, can be seen as a combination of L-systems and graph grammars. More precisely, we integrated the operator approach to parallel graph grammars within the mechanism of parallel single-pushout derivations, and showed which special connection transformations are required for an embedding of L-systems which is equivalent to the original L-system formalism. As a consequence, relational growth grammars inherit the proved strength of L-systems for plant modelling. But being based on graphs and graph grammars, they are able to represent even more structures and their dynamics in a natural, convenient way as it was shown by the diverse collection of examples.

The theory behind relational growth grammars is not new. The single-pushout approach is known since more than a decade, the operator approach even longer. The motivation for the development of the single-pushout approach was its dispensation with implicit gluing conditions, this yields (from a practical point of view) a very convenient and efficient, but nevertheless clean and well-founded algebraic derivation mechanism. The development of the parallel variant of the operator approach was in the wake of the enthusiasm for L-systems, but has not been used afterwards for the modelling of biological systems. In this work we revitalized the operator approach by showing how it can be used as an add-on to parallel single-pushout derivations. The precise definition uses two-level derivations, which were first studied in

the context of amalgamation, but turn out to be also very suitable for the inclusion of the operator approach.

Relational growth grammars introduce dynamic right-hand sides, i. e., an instantiation of actual rules on the basis of given matches. This is exclusively motivated by practical considerations as it reduces the necessary amount of rules in cases where, depending on some conditions like attribute values, only the right-hand sides differ. On the other hand, this expressive power reduces the strength of the formalism from a theoretical point of view, i. e., less can be proved when dynamic right-hand sides are used. But the definition of relational growth grammars was always guided by the usefulness as a basis for functional-structural plant modelling with the focus on simulation, not on proving properties. Nevertheless, using the formal description of a model to prove general statements about its development is an interesting challenge. An example thereof in the context of botany and L-systems is [158].

11.2 The XL Programming Language

The main practical result of this work is the design and implementation of the XL programming language. Although relational growth grammars guided the design of this language by their requirements, the XL programming language is completely independent of relational growth grammars or any other concrete rule-based formalism. It is only the abstract rule-based paradigm which is inherent in the XL programming language, namely by the dedicated rule syntax. Its semantics is sufficiently abstract to allow for a variety of concrete rule-based approaches, including but not restricted to algebraic graph rewriting or vertex-vertex algebras. In order to achieve this high level of abstraction, we did not reinvent the wheel, but built on the well-known technique of operator overloading. Likewise, for the whole programming language we used the Java programming language as a well-designed foundation. This means that we do not only inherit the strength of the Java programming language, but also of its numerous run-time libraries. Extending a widely used and clean programming language, we felt committed to do this in a careful way, and we believe that this was achieved to a large extent.

An important step in the development of the XL programming language was the split between left-hand sides and right-hand sides of rules. Left-hand sides are queries, right-hand sides are production statements, and both queries and production statements can also be used on their own right. This is of special importance in the case of queries as these can be used to evaluate arbitrary context in a convenient way. The split also follows the principle of separation of concerns and helps to develop, implement and optimize both parts separately.

For left-hand sides and queries, a versatile textual syntax was designed. This does not only cover the specification of conventional graph patterns, but also of applications conditions and of path expressions composed of, e. g.,

method invocations and transitive closures, and of optional patterns. The expressiveness can be seen in the Ludo example where the legal movement of a pawn was implemented by a single rule. Besides expressiveness, also the issue of generality was an important design goal for queries, and it was addressed by means of the XL interfaces through which the pattern matching algorithm accesses the actual data structure. The implementations for commercial 3D modellers as well as the implementation for the minimalistic Sierpinski graph document the fitness of the mechanism for a variety of data structures.

The pattern matching algorithm of the run-time system is implemented in a recursive way. This allows the usage of normal and generator expressions, which yield values via call-back consumer interfaces, in patterns, and it also makes possible a natural implementation of transitive closures. On the other hand, when transitive closures are applied to deep structures, the deep recursion may lead to a stack overflow. The usual solution of such a problem by a non-recursive implementation with the help of a stack-like data structure is not compatible with the mechanism of generator expressions, so up to now there is no solution except for the allocation of a sufficiently large stack space.

The order in which the matching algorithm processes the components of a pattern has a major effect on its efficiency with respect to time. Therefore, an optimal order is computed based on a heuristic cost model. This computation is performed only once, namely when the pattern is used for the first time for matching, and it does not yet use information about the graph in which matches shall be found. This could clearly be improved by a mechanism which detects when the structure of the graph has changed enough so that a computation of a new order pays off.

For right-hand sides and production statements, even less properties of the underlying data structure are prescribed by the XL programming language. There is no data interface for some standardized modifications, only a producer is used which may offer an arbitrary set of modifications by the implementation of operator methods. Thus the definition of production statements is rather minimal, but exactly this allows a rather maximal freedom in the possible semantics of these statements. This was shown by the ease and usability of the implementation of parallel vertex-vertex algebras, which originally was not intended when designing the XL programming language.

From the examples in Sect. 10.5 on page 296 we can see that the aim of this work, the design and implementation of a language for functional-structural plant modelling, was achieved to a satisfactory extent. Utilizing the possibility of the XL programming language to operate on graphs, we can concisely represent plant structure and its dynamics in the framework of graphs. This is of course a more natural representation than words as it is implicit within L-systems. The advantages can best be seen at examples like the ABC model where a regulatory network is represented by a graph, the ant model with its grid structure of the world and its representation of memory by edges, and the beech model where the possibility to navigate within the graph in an arbitrary direction is utilized in order to implement both a basipetal and

an acropetal transport through the whole structure in a single step. But even if we do not make any use of the representation of the topology by a true graph, there are still advantages: Nodes are objects in the sense of object-oriented programming. Their classes can be equipped with methods and attributes, and they define an inheritance hierarchy of which one can take advantage in modelling. Equally important, nodes have an identity by which they can be addressed. This allows to reference them globally at any place in the model, regardless of a "cursor" in the structure like the current derivation position of an L-system interpreter. Contrary to L-systems, nodes of the graph may not only be created or deleted like L-system symbols, but may also be kept and modified with respect to their parameters.

Graph queries can be seen as an extension of local context of L-systems. They greatly simplify the specification of functional-structural models as they allow to search for nodes that fulfil certain conditions in an arbitrarily large context (namely, the whole graph). As a consequence, every node can have an influence on any other node within a single rewriting step, whereas L-systems restrict this influence to a finite local context (or, depending on direction of derivation, to symbols to either the left or right [91]). Thus, queries can be used to implement arbitrary environmental interactions. An example thereof is shown on page 159 where a globally-sensitive function of the GROGRA software is implemented by a combination of a query expression and an aggregate method.

In general, when the XL interfaces are implemented according to relational growth grammars, we have a natural translation of nearly any type of L-systems presented in Chap. 3 to the XL programming language. *Stochastic L-systems* are emulated by using some pseudorandom methods. *Context-sensitive L-systems* are covered by the possibility to specify arbitrary graph patterns on left-hand sides. The control flow of *table L-systems* is a special case of the control flow of imperative programming. *Pseudo L-systems* allow more than one symbol on the left-hand side of productions, this corresponds to graph rules with more than a single node as left-hand sides and the nondeterministic parallel derivation mode. However note that there is a difference as the definition of pseudo L-systems in [153] includes processing of the current word from left to right, which is not possible in the graph setting as there is no intrinsic ordering from left to right. *Parametric L-systems* perfectly fit into the framework of typed attributed graph grammars and object-oriented programming. *Interpretive productions* can be obtained by special graph structures and derivation modes, but the XL programming language also provides the mechanism of instantiation rules which generally is preferable. The speciality of *growth grammars*, a dedicated set of globally sensitive functions, can be obtained by implementing the sensitive functions on the basis of queries, aggregate methods and, where necessary, some general elementary sensitive functions. Likewise, *environmentally sensitive L-systems* only require elementary sensitive functions related to the local coordinate system of a node. The general mechanism of *open L-systems*, i. e., the communication between an

L-system and some external environment, is also a special case of the possibilities of the XL programming language: we divide a model into a rule-based structural part, generalizing the notion of L-systems, and a functional part for the environment which is typically implemented following the imperative paradigm. However, the concrete mechanism of open L-systems based on the cpfg software, where environments are defined by library files and a binary interface, is not supported. Anyhow, this mechanism is rather a legacy of the strict separation between structural and functional parts, which to resolve is the aim of this work.

Differential L-systems are not yet supported to a satisfactory extent: while it is possible to incorporate numerical solvers for differential equations by computing the required data and invoking some methods, a more direct integration into rules is desirable. For example, for a system of differential equations with respect to time one may want to specify the current rates for some quantities within a set of rules, and a sophisticated solver then integrates the equations until some event like a reached threshold happens. By specifying the rates within rules, we automatically take into account the current structure and its topology, which may change at discrete points in time, thus arriving at a dynamical system with dynamical structure [66] on the basis of differential equations and a precise solver.

In the field of plant modelling, there currently is to our knowledge only one comparable approach, the L+C programming language as part of the L-Studio software (see Sect. 3.14 on page 33 and [91, 92, 159]). It is an implementation of parts of the proposal for an L-system based plant modelling language called L [160]. As mentioned in Sect. 3.14 on page 33, the L+C programming language uses a source-code preprocessor which translates the special production syntax of L+C into valid C++ code. Therefore, where the L+C programming language allows imperative blocks, it inherits the full strength of C++, and the comparison between the L+C and XL programming languages amounts to a comparison between C++ and Java. But the locations where imperative blocks are allowed are very restricted: following the traditional cpfg syntax of L-systems with imperative programming statements (Sect. 3.10 on page 29), blocks are allowed only as part of right-hand sides and as the special blocks **Start**, **StartEach**, **End**, **Each** which are executed at distinct points of the derivation process. This means that the overall structure adheres to the declarative, rule-based character at the expense of versatility concerning control of rule application. In the L+C programming language, this control is only possible via a mechanism similar to table L-systems [159]. Contrary, the XL programming language reverses the overall application logic: this is governed by the imperative part which may choose to execute a set of rules. Thus, loops, switches between rule sets, rule application as long as possible and any other control of rule application is possible.

Both the L+C and XL programming languages allow a dynamic construction of the successor on right-hand sides. But here, the situation concerning the declarative or imperative character is inverted. For the L+C programming

language, a right-hand side is an imperative block, and the successor has to be specified by special **produce** and **nproduce** statements. For the XL programming language, a right-hand side specifies the successor immediately, and imperative code may be inserted in braces. As an example, the L+C production

```
X(t): {
    float x = pow(t, exp);
    nproduce F(1) SB();
    if ((t & 1) == 0)
        nproduce Left(60);
    else
        nproduce Right(60);
    produce Flower((int) x) EB() X(t+1);
}
```

translated to XL code looks like

```
X(t) ==> {float x = Math.pow(t, exp);}
    F(1) [  if ((t & 1) == 0) (RU(60)) else (RU(-60))
            Flower((int) x)
         ]
    X(t+1);
```

In our opinion, the syntax of the XL programming language is more natural in this respect, especially when considering simple rules.

For functional components of plant models, local and global context is of great importance. The XL programming language allows the convenient consideration of context both directly on the left-hand side and by the usage of queries, possibly combined with aggregate methods. The L+C programming language follows the traditional cpfg syntax, which only allows local context specified on the left-hand side, but adds a mechanism for new context to enable a fast unidirectional information transfer (Sect. 3.14 on page 35). Adhering to the traditional notion of context has drawbacks. We can only specify single patterns for left and right context, which fails, for example, when we need several unrelated entities like the nearest internode and the nearest carrier of some substance. We cannot specify context of an unknown size like all internode children. The XL programming language provides solutions to these problems: left-hand sides may contain several context patterns, and we may also use queries on the right-hand side to process context of unknown size, typically in combination with aggregate methods. E. g., think of the computation of the total cross section of all internode children. The original specification of the language L also proposed two solutions to the first problem by the alternatives

```
if (lcontext(A(s, t))) {...}
if (A(s, t) < self) {...}
```

This is a simple kind of query expression, but is not part of the L+C programming language.

But the major difference between the XL and L+C programming languages is definitely the range of supported data structures. The L+C programming language makes an explicit reference to a fixed data structure, namely a linear sequence of modules which may be parameterized by C++-structures. Contrary, the XL programming language defines the data structure through its interfaces. This allows linear sequences, trees, graphs, graph rotation systems, nearly any custom data structure as the target of rules, with instances of Java classes as basic components, and the performed modifications may conform to a variety of formalisms like parallel or sequential graph rewriting or vertex-vertex algebras. The typical data structure is a graph, this enables to represent plant topology in a natural way. It also makes possible a flow of information along arbitrary paths in the structure, while the processing of the current word in the L+C programming language only allows local context to be considered, plus the part of the word either to the left or to the right, depending on derivation direction.

Components of the structure being instances of Java classes, we have a convenient inclusion of the object-oriented principles of inheritance and polymorphism. Therefore, it is possible to specify common behaviour of a set of related classes by a common superclass, and to specialize the behaviour in its subclasses. We used this technique in several examples like the beech model with the class `Organ` being the superclass of all organ classes. Such a concept of inheritance is not yet available in the L+C programming language, but it is already mentioned in [91] and also in the specification of the language L [160] as a desirable feature.

Concerning general graph rewriting software, there exists a diverse range of tools with different strengths and weaknesses, depending on the specific aims of the tools. The AGTIVE '07 tool contest (Sect. 10.8 on page 347) with its three case studies, the Sierpinski triangle, the Ludo game and the UML-to-CSP conversion, was a good opportunity to compare a wide range of tools along several dimensions of features. Given that virtual plant structures quickly become large, efficiency concerning time and space is crucial. The Sierpinski case study provides a benchmark to test this efficiency, and implementations based on the XL programming language turn out to be among the fastest and most lightweight solutions, particularly if a specially tailored graph representation was used. The Ludo case study also shows that the XL implementation is relatively fast, but here the focus is rather on the expressiveness of the tools – how concise is the representation of the game rules? Taking advantage of the possibility of transitive closures and optional patterns, the XL implementation is short and concise. At the same time, it provides a nice 3D visualization using the facilities of GroIMP. The solution of the UML-to-CSP case study shows the usefulness of the XL programming language and GroIMP to implement a translator for graph languages. It utilizes the possibility to have different derivation modes in succession. But this solution also shows some shortcomings of the XL programming language and of the implementations of its interfaces. Namely, some other tools allow to check the

compliance of graphs with their graph model (i. e., their type graph), and there are also tools like AGG which can analyse the transformations themselves to detect dependences of the final result on the order of sequential production application. The first issue could be solved at the level of implementations of the XL interfaces by the inclusion of validation of graphs with respect to their model. But the second issue is intrinsic to the design of the XL programming language. Rules are just special statements which are executed when the control flow reaches them. At no point in time there is knowledge of all rules which comprise a single rewriting step, and even if we know a single rule, there is no knowledge of the structure of its right-hand side as this is a dynamic property, to be constructed by the producer on execution. This is yet another manifestation of the typical trade-off between a rigorous framework which allows proving a lot of properties and a practically convenient framework which allows to express complex behaviour by only a few statements in compact notation. The XL programming language clearly belongs to the second kind.

Concerning efficiency, the graph transformation tool GrGen.NET is comparable to the XL programming language and, thus, can principally and practically also be used to describe plant growth. It also provides a versatile means to specify graph patterns, application conditions, right-hand sides and computation of new attribute values. But in the current version, optional patterns or transitive closures are not supported, and there is no direct embedding of rules in an outer imperative programming language. Theses features are intended for a future version of GrGen.NET. Also a parallel derivation mode and a connection mechanism to simulate L-systems are not built into GrGen.NET.

Of course, efficiency can and should also be compared with L-system software. Due to the simple string representation, data management in L-system software is easy and fast, while graph-based tools have some overhead in this respect. Therefore, we expect that L-system software can outperform graph-based tools if most of the time is spent in changing the data structure. This is the case for pure structural models without complex computations like the following RGG code for a ternary branching:

```
const float ANGLE = 60;
const float FACTOR = 0.6;

module X(float s);

protected void init()[
    Axiom ==> X(1);
]

public void run() [
    X(s) ==> F(s) [RU(ANGLE) X(s*FACTOR)] [RU(-ANGLE) X(s*FACTOR)] X(s);
]
```

On an Intel Pentium 4 CPU with 2.8 GHz, 11 steps take about 4.76 seconds and create a final graph of 442,866 nodes. A corresponding implementation in

the L+C programming language, executed within L-Studio, takes only about 0.34 seconds, while the equivalent growth grammar, executed within GROGRA, needs about 6.74 seconds (note that GROGRA is an interpreter, while L+C and XL code is compiled). Similar to the Sierpinski grammar, we can reduce the computation time by using a custom graph based on the simple implementation of the XL interfaces. With such a custom graph, the 11 steps require only about 1.17 seconds. Clearly, this is still less efficient than the L+C solution, but we have to keep in mind that graphs are more complex to handle than strings. This is the price we have to pay for the gain in expressiveness and versatility. On the other hand, really complex functional-structural plant models are not dominated by the time for the creation of structure, but by the time for computations within existing structure, for example a radiation model as in the example of a spruce-beech stand in Sect. 10.5.6 on page 326. Then the overall efficiency mainly depends on the efficiency of these algorithms.

11.3 Outlook

Although we have shown the usefulness of the formalism of relational growth grammars and of the XL programming language for functional-structural plant modelling at several examples, it is now the time to develop further, more elaborated functional-structural plant models in order to validate the usefulness. This should be done by different research groups with different aims, so that a variety of dimensions of the formalism and the language are explored, possibly leading to suggestions for improvements or new features. Also the applicability for other purposes than plant modelling could be further investigated. In conjunction with the GroIMP software, this seems to be most promising in the field of algorithmic 3D modelling, the architectural examples being first hints into this direction.

One useful extension, to which some of the examples already point, is the inclusion of differential equations. A precise functional part of a plant model typically contains systems of differential equations, e. g., for transport or metabolism. A built-in feature to specify and solve these equations in a numerically stable way is desirable. It should use the ideas of differential L-systems, especially the idea that certain thresholds can be defined which when reached trigger some discrete action like the creation of a new organ when some hormone concentration is reached (see also the ABC model of flower morphogenesis, Sect. 10.5.1 on page 296). Ideally, the existing features of the XL programming language like operator overloading and properties already suffice to implement such an extension.

The ease of the implementation of a parallel vertex-vertex algebra for the specification of the dynamics of polygonal surfaces gives rise to the question if and how this can be extended to polyhedral volumes. By its very nature, this is not trivial, especially when the extension shall still be practically useful and convenient for plant modelling, e. g., for the modelling of tissues represented

by polyhedral cells. There exist some formalisms, but none has established itself to an extent comparable with L-systems for modelling of plant topology.

Another interesting topic for future research is the specification of rules by graphs, so that we may have meta-rules transforming rules. A typical application is the modelling of evolution like in genetic algorithms. The implementation should not require any extension of the XL programming language, but the specification of special node classes representing rules, and the implementation of an algorithm which takes graphs of such nodes as input and modifies the structure accordingly. This algorithm could re-use the built-in matching algorithm of the run-time system of the XL programming language.

The matching algorithm itself is also a target for improvement. A substantial enhancement concerning efficiency would be the dynamic generation of a new search plan if the graph has changed sufficiently. The problem is to find a good definition of a "sufficient change" so that the required time for the generation of the new search plan is less than the saved time for matching. Further improvements could be achieved by caching occurrences of frequently used subpatterns, similar to the extent lists of GroIMP which store all occurrences of a given node type.

A further possibility to speed up model execution is to compute in parallel. This can be done either on a single multi-processor machine, or on a network of several connected machines. The radiation model already makes use of multiple processors, and we started some investigations to use multiple processors for rule execution. In principle, the framework is well-suited for parallel computation: if modifications to the structure are done exclusively via modification queues, the main part of model execution is to fill these queues with corresponding entries, but it does not change the structure. Thus, we can construct one queue for each processor of each connected machine, and combine these partial queues to a single queue (i. e., a parallel production) at the end of a step when the parallel derivation is applied. The necessary communication amounts to the transfer of partial queues and resulting graph changes. The second part, the synchronization of several GroIMP instances connected over the network by transferring graph changes, is already implemented in the GroIMP system, but not yet the first part and a mechanism to partition rule execution among available processors.

Finally, the nature of the XL programming language as an extension of the Java programming language could be exploited in a better way and with a broader reach by an integration into popular development tools like the Java compiler `javac` or the Eclipse platform. When this work was started, `javac` was a closed-source program, and the Eclipse platform in its infancy, so that a usage of them as basis for the XL compiler was not considered. Nowadays, both tools are established open-source projects, and we could extend their existing compilers by features of the XL programming language. Through the Kitchen Sink Language project [186], which provides an official central point for suggestions of new language features, this could even have some influence on the evolution of the Java programming language.

Acknowledgements

The main part of this research was done at the Chair for Graphics Systems at the Brandenburg Technical University. I thank my adviser Prof. Dr. Winfried Kurth for giving me a very interesting and challenging task for my doctoral thesis. His comprehensive and precise support and suggestions helped a lot in carrying out this task.

Prof. Dr. Winfried Kurth used his contacts to make possible my stay of one and a half year at the institute for Ecoinformatics, Biometrics and Forest Growth of the University of Göttingen. I am very grateful to Prof. Dr. Dr. h.c. Branislav Sloboda, the director of the institute, for kindly affiliating me as a guest researcher and providing the necessary infrastructure.

I also want to thank my colleagues in Cottbus and Göttingen for the nice working atmosphere and fruitful discussions. Dr. Gerhard Buck-Sorlin provided his profound biological expertise without which the presented work would not have been possible. As a GroIMP and XL user from day one, his feedback was a major contribution to the usability of the software. Reinhard Hemmerling was a constant source for new ideas. He included operator overloading in the XL programming language, which then became the foundation for right-hand sides and their versatility. He also made possible my stay in Göttingen by a swap of our positions. Jan Dérer improved the software by a lot of useful analysis functions, and he established new contacts. Dr. Dirk Lanwert as an expert user of GroIMP and XL made a lot of suggestions and bug reports. His knowledge of tree growth helped a lot to implement the beech model. Katarína Smoleňová developed several plant models and created nice visualizations.

A lot of students have contributed to the presented work by their theses. I thank Udo Bischof, Benno Burema, Birka Fonkeng, Bernd Gräber, Christian Groer, Michael Henke, René Herzog, Oliver Huras, Thomas Huwe, Daniel Klöck, Ralf Kopsch, Ralf Kruse, Sandy Lobe, Uwe Mannl, Mathias Radicke, Stephan Rogge, Sören Schneider, Hagen Steidelmüller, Michael Tauer, Stefan Willenbacher and Dexu Zhao for their work.

As a tutor of a course in Computer Graphics and Software Engineering at BTU Cottbus, I got helpful insights into the field of software design from Prof. Dr. Claus Lewerentz. These definitely improved the design of GroIMP and XL, and will also help in the future.

I thank Prof. Dr. Hans-Jörg Kreowski, head of the research group Theoretical Computer Science at the University of Bremen, for his interest in my work and for inviting me to Bremen. With him and his research assistant Caroline von Totth I had a fruitful discussion about this thesis.

The more complex models presented in this work would not have been possible without the computing power which Dr. Reinhold Meyer from the institute for Ecoinformatics, Biometrics and Forest Growth made available to me. I also want to thank Andreas Hotho and Jörn Dreyer from the Knowledge and Data Engineering Group of the University of Kassel for kindly providing me with an account for their powerful computer.

Part of this work was funded by the Deutsche Forschungsgemeinschaft under grant Ku 847/5 and Ku 847/6-1 in the framework of the research group "Virtual Crops".

Last but not least, I would like to thank Anja and Hannes for their support and patience during the last years. Sharing my home life with them is an important pleasure.

A

The Modelling Platform GroIMP

In this appendix, we briefly describe the modelling platform GroIMP – the growth-grammar related interactive modelling platform. The design and implementation of the XL programming language and the modelling platform GroIMP happened in parallel. In fact, in the early stage of development both were inseparably tied together, but by the introduction of the XL interfaces, this coupling was removed. Now it is only the RGG plug-in of GroIMP which establishes an explicit link between both parts, but there are several further "soft" links implied by a convenient and useful implementation of the XL interfaces, for example the requirement to be able to efficiently handle very large graphs (more than a million of nodes, say) which may be very deep (like a linear chain).

GroIMP being composed of more than 3,000 classes, we cannot describe everything in detail in this appendix. We rather give an overview of the application, its structure and the most important features from a user's perspective. For more information, we refer the user to the online help, the API documentation and additional material available at [101].

A.1 Overview

GroIMP is designed as an integrated platform which incorporates modelling, visualization and interaction. It exhibits several features which make it suitable for functional-structural plant modelling, but also for other tasks:

- The "modelling backbone" consists in the XL programming language. It is fully integrated, e. g., the source code is edited in an integrated text editor and automatically compiled by the XL compiler. Errors reported by the compiler are shown in a message panel and contain hypertext links to their source code locations.
- GroIMP provides a complete set of 3D-geometric classes for modelling and visualization. This includes turtle commands, primitives like spheres,

cones and boxes, height fields, spline surfaces, and polygon meshes. Spline surfaces are in fact NURBS surfaces and can be constructed by several techniques, among them surfaces of revolution and swept NURBS (generalized cylinders) [149].

- In addition, GroIMP provides a *shader system* for the definition of 3D shaders. Shaders can be built by combining image textures and procedural textures in a flexible, data-flow oriented way as it is known from up-to-date 3D modellers like Maya [5].
- The outcome of a model is visualized within GroIMP by several options including a real-time display based on OpenGL and the built-in raytracer *Twilight*.
- In the visual representation of the model output, users can interact with the dynamics of the model, e. g., by selecting, modifying or deleting elements.
- GroIMP contains a 2D view on graphs that shows their structure. A set of layout algorithms can be applied to arrange the nodes in this view [60, 199].

GroIMP is open-source software implemented in Java; it is licensed under the terms of the General Public License, version 3. The latest version and information can be found at the web page `http://www.grogra.de/`. Figure A.1 shows a screenshot displaying the Ludo example of Sect. 10.8.1 on page 347.

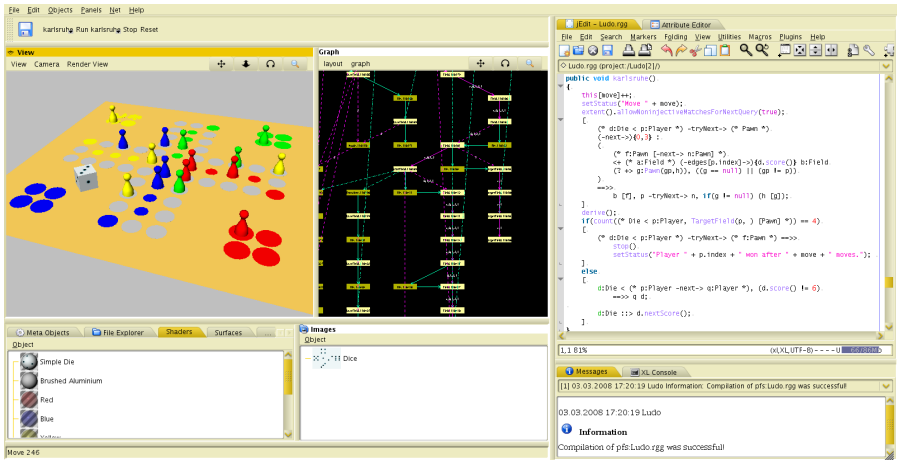


Figure A.1. Screenshot of GroIMP window displaying the OpenGL-based 3D view, a 2D view on the graph structure, the source code editor, the Shader explorer with a list of shaders, the Image explorer with a single image, and the message panel

A.2 Plug-In Architecture

At a global level, GroIMP has a plug-in architecture very similar to the Eclipse platform [40]. A relatively small core part loads a list of plug-ins on start-up of the application. Each plug-in is described by an XML file `plugin.xml` which defines, among others, its capabilities and a list of other plug-ins that are prerequisites. Capabilities are represented as a hierarchical structure, addressed by names, which is merged for all plug-ins into a single registry. Each plug-in is loaded by its own class loader which ensures that, besides the own classes of the plug-in, only classes from plug-in prerequisites can be loaded. By carefully dividing the functionality into different plug-ins, this enforces a clean design without cyclic dependences at the global level of plug-ins.

In the source distribution of GroIMP 0.9.8.1, the core is split into four projects:

- The project XL-Core contains the basic packages `de.grogra.xl.lang` and `de.grogra.xl.util`. The specification of the XL programming language relies on all types of `de.grogra.xl.lang`, while `de.grogra.xl.util` contains several basic utility classes.
- Utilities contains a miscellaneous set of utility classes.
- Graph defines both a graph interface and an implementation thereof, see Sect. A.3 on page 379.
- Platform-Core is the actual core of GroIMP. It defines the classes for the registry and the main class `de.grogra.pf.boot.Main` which starts the application.

The version 0.9.8.1 of GroIMP contains the following plug-ins which are also shown in Fig. A.2 on page 380 with their dependences.

- Platform is the basic plug-in of GroIMP and is referenced (directly or indirectly) by most other plug-ins. It contains an abstraction of a graphical user interface around the interface `Workbench`.
- Platform-Swing implements the user interface for the standard Swing toolkit of Java. jEdit replaces the default editor of Swing by the sophisticated editor jEdit.
- Vecmath contains an implementation of the `javax.vecmath` library and an implementation of 3D geometry (primitives, polygon meshes, constructive solid geometry, octree) with the focus on ray-object intersections for the purpose of ray-tracing.
- Math defines interfaces and implementations for simple functions, vertex lists, B-spline curves and surfaces. This includes non-uniform rational B-splines, i. e., NURBS [149]. A lot of algorithms of [149] like the computation of swept surfaces are also provided.
- IMP adds facilities to display and edit graphs as defined by the Graph project from above. Furthermore, it defines support for images, fonts and some other data types. Finally, it contains a simple implementation of an

HTTP server. This is an interesting feature and enables GroIMP models to serve as dynamic content providers for web applications. For an example, see Sect. 10.5.8 on page 330.

- TexGen contains several texture generators, i. e., algorithms for the creation of synthetic images.
- IMP-2D provides node classes for 2D geometry. Such a 2D scene graph can be displayed in a 2D view, but there is also the option to display a 2D view on the topology of an arbitrary graph.
- IMP-3D plays a central role as we can see in Fig. A.2 on page 380. It defines node classes for all kinds of 3D geometry from primitives like points, spheres and boxes to complex objects like NURBS surfaces and polygon meshes. Cameras, light sources and sky objects are defined, and physically valid shaders can be constructed from procedural textures and images. This plug-in also provides a 3D view with the option to use OpenGL for rendering, and manipulation tools for translation, rotation and scaling.
- Raytracer is used by IMP-3D to integrate the raytracer *Twilight* (Sect. A.7.1 on page 392), which is a subproject of the whole GroIMP project but independent of the GroIMP application. This raytracer implements both a conventional ray-tracer and a path-tracer based on [195]. For the representation of geometry, it relies on the plug-in Vecmath. Its algorithms are also used for the radiation model outlined in Sect. B.14 on page 412.
- Sunshine is an implementation of a ray-tracer which uses the GPU of the graphics card [81].
- POV-Ray, DXF, X3D, PDB and CPFG define import and export filters for various (mostly 3D) data formats. POV-Ray also embeds the POV-Ray ray-tracer so that it can directly render in the 3D view.
- Billboard facilitates the creation of snapshots from different perspectives for the usage in billboards, see Sect. 10.5.9 on page 332.
- XL contains the classes and interfaces of Chap. 6 which are not already contained in XL-Core, i. e., the packages `de.grogra.xl.query`, `de.grogra.xl.property` and `de.grogra.xl.modules`.
- XL-VMX implements the extension of the virtual machine as presented in Sect. 8.4 on page 212.
- XL-Compiler contains the compiler of Chap. 8.
- Grammar is used by XL-Compiler, it provides a general-purpose lexical analyser as described in Sect. 8.1 on page 207.
- XL-Impl contains the base implementation of the XL interfaces, see Chap. 9.
- RGG is the most important plug-in for rule-based modelling with GroIMP as it defines a versatile implementation of the XL interfaces for the graph of GroIMP and with the semantics of relational growth grammars. This plug-in is described in detail in Chap. B.
- 3D-CS contains a construction set similar to Xfrog [121, 37] which was designed as part of a Diploma thesis [77].
- There are also some further plug-ins without source code or with only some helper classes. RGG-Tutorial is a short tutorial to introduce the usage

of relational growth grammars to the novice. API-Doc provides a single menu entry which opens the API documentation of GroIMP (generated by `javadoc`). Examples also provides a single menu entry, this shows a list of examples. Platform-Swing-LookAndFeels contains a collection of nice non-standard look-and-feels for the Swing toolkit.

A.3 Graph Component

The Graph component of the core classes contains three packages. The package `de.grogra.graph` defines an abstraction of graphs. `de.grogra.persistence` provides a mechanism to manage objects and their attributes including storage, reading in, logging of changes, and grouping several changes in single transactions which can be undone. `de.grogra.graph.impl` uses both packages and implements an efficient graph representation which is also used for the implementation of relational growth grammars of the RGG plug-in.

A.3.1 Graph Interface

The graph interface in the package `de.grogra.graph` is defined around the interface `Graph`. The latter has a lot of methods to query information about the topology and attributes of nodes and edges, which are passed as parameters. Some methods are shown in the class diagram in Fig. A.3 on page 381. For example, a loop over all edges `e` of a node `n` of a graph `g` is written as

```
for (Object e = g.getFirstEdge(n); e != null; e = g.getNextEdge(e, n)) {
    ... // do something with edge e
}
```

But there is also the method `accept` which passes the topology of the graph to the specified `Visitor` by invoking the callback methods of the latter. The mechanism follows the hierarchical visitor pattern.

An important property of the interface is that it does not restrict the actual types of nodes and edges: any `Object` can be used as long as the methods of the `Graph` can handle the requested operations. The converse solution would be to define interfaces for nodes and edges with topological operations directly provided by them. But then the actually used classes for nodes and edges are required to implement these interfaces, which excludes the possibility of externally defined graph structures. In our setting, such structures can be integrated easily by the implementation of the `Graph` interface which serves as a facade (a design pattern presented in [62]). A similar approach (but only for trees) is taken by the interface `javax.swing.tree.TreeModel` of the Swing toolkit.

The method `getAttributes` returns, for a given node or edge, an array of its attributes. Thus, the interface does not yet restrict edges to be of a simple

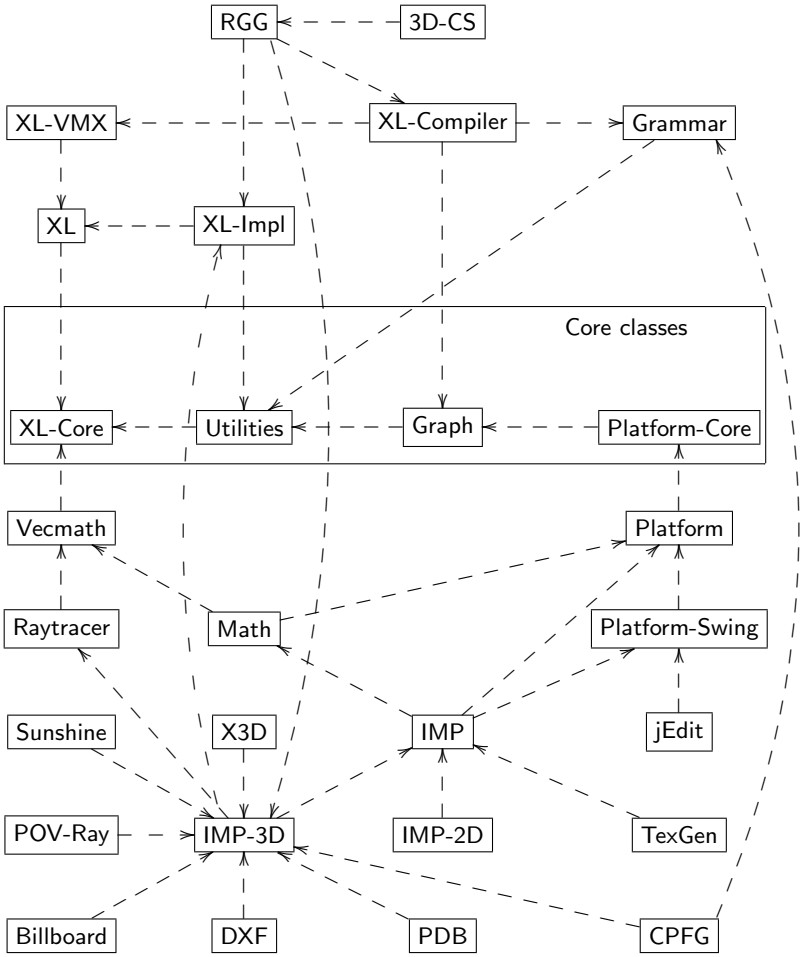


Figure A.2. Core classes and plug-ins of GroIMP and their dependences. The plug-ins above the core classes are related to relational growth grammars and the XL programming language, the plug-ins below the core classes constitute the basic part of GroIMP including 3D modelling support. The dependence of RGG on IMP-3D establishes the link between relational growth grammars and 3D scene graphs. There are two further dependences between both otherwise separated parts: IMP-3D depends on XL-Impl because classes for primitives contain user-defined parameterized patterns (so that we may write, e. g., `Sphere(r) ==> Sphere(2*r);`); CPFG depends on Grammar since it uses the general-purpose lexical analyser of the latter (but note that Grammar is independent of the XL plug-ins so that its assignment to the upper part is arbitrary). The only questionable dependence is from IMP to Platform-Swing. This is because the abstraction of the graphical user interface in Platform does not define components for (2D or 3D) display of graphs, which is therefore added in IMP and at the same time implemented for the Swing toolkit.

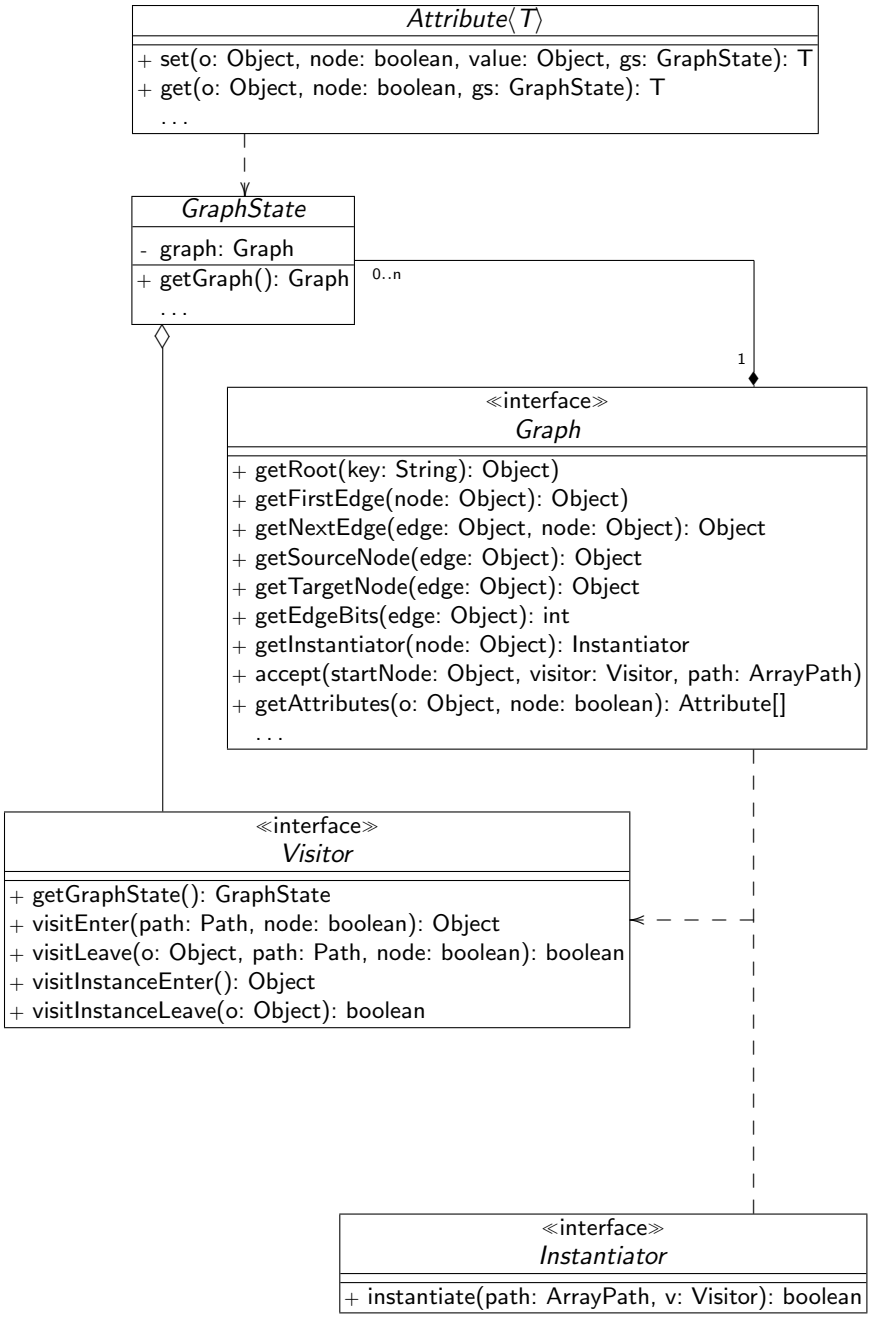
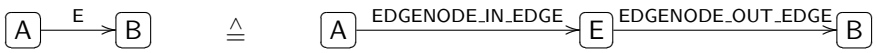


Figure A.3. Class diagram of graph interface

non-attributed nature. Attribute values can be read and written by corresponding getter- and setter-methods. These methods require an additional `GraphState` as argument, which maintains per-thread state information of a graph. This state is most important for object instancing as it stores the complete information about the currently instantiated nodes and edges and their attributes.

Object instancing is triggered within the method `accept`, i. e., when a `Graph` passes its structure to a `Visitor`: if, for a visited node, the method `getInstantiator` returns an `Instantiator`, this is used to (temporarily) instantiate a structure by invoking the corresponding methods on the `Visitor`. As a consequence, at no given point in time the complete instantiated structure exists, we rather have only information about the path from the beginning of the instantiation to the currently instantiated object. (An exception is the case where the instantiated structure is a linear chain, then the path to the last object contains information about the complete chain.)

There is a single feature of the graph interface which is influenced by its later usage for the graph of GroIMP and the base implementation of the XL interfaces (Sect. 9.1.1 on page 235): each edge is assumed to have a set of edge bits which is returned by the method `getEdgeBits` of the graph as a single **int** value. Following the discussion in Sect. 9.1.1, these bits can be used to encode up to 32 individual non-attributed edges or, and this is used here, 24 such edges and a single additional edge whose type is encoded by the remaining 8 bits together. The `Graph` interface defines constants for edges of type successor, branch and several further ones. The special types `EDGENODE_IN_EDGE` and `EDGENODE_OUT_EDGE` are used when a node shall play the role of an edge: such a node has a single incoming edge of type `EDGENODE_IN_EDGE`, whose source node is then considered as the source node of the compound edge, and a single outgoing edge of type `EDGENODE_OUT_EDGE`, whose target node is considered as the target node of the compound edge:



This trick has to be used when the graph model of RGG and/or its implementation is not sufficient, i. e., when attributed edges or parallel edges of the same type are needed, or when one needs more edge types than can be represented in the 32 edge bits.

In principle, graphs may have an arbitrary topology. However, when they are interpreted as 2D or 3D scene graphs, there would be a problem if a node can be reached by more than one path from the root of the scene graph: each such path defines its own local coordinate system by collecting all coordinate transformations of the objects along the path, and the resulting local coordinate systems for different paths might not coincide. Therefore, there is a special convention for scene graphs which has to be obeyed: only those objects are visible which can be reached from the root of the scene graph by traversing successor or branch edges in forward direction, and for any visible

object, there must not exist two such paths. In other words, the subgraph spanned by successor and branch edges when starting at the root has to be a tree. There are no restrictions concerning other edge types or branch or successor edges between non-visible nodes. Using this convention, we can not only define the local coordinate system for each visible object without ambiguity, but also general derived attributes, i. e., attributes which are defined recursively based on the value of the parent and some local attributes. E. g., this is used for the turtle state (Sect. B.12 on page 406).

A.3.2 Management of Objects, Attributes and Changes

The graph interface does not provide a means for persistence, i. e., for reading from and writing graphs to durable storage like files. In principle, the standard serialization mechanism of Java can be used for that. However, for the purposes of GroIMP a more sophisticated mechanism is required which does not only handle the serialization of graphs into byte streams, but also provides the following features:

- There exists knowledge about the attributes and their types and values for any node.
- This knowledge is fine-grained in the sense that we also know how non-primitive attribute values are composed. E. g., an attribute for a list of locations in 3D space is composed of an array, and each array component is composed of three floating-point values.
- Changes to the structure and attribute values are noticed and may be logged to a protocol. Access to attribute values is fine-grained so that we may modify only a component of an attribute value, and then only this small change is logged. E. g., we only change the z-component of the component with index 1 of an attribute for a list of 3D locations. Together with the previous feature this allows a convenient implementation of XL properties including subproperties and component properties (Sect. 6.10 on page 174).
- Change logs may be grouped to transactions which may be undone.
- Transactions may be written to streams so that they can be exchanged with other instances of the same graph. Think of several GroIMP instances, linked over a network, which shall work on the same graph. This has the potential of applying RGG rules in parallel on different machines.

Java Data Objects (JDO, [184]) were specified with similar and a lot of further goals in mind, but they do not provide a means for fine-grained access of attribute values, and they are not able to handle edges in the way we need it, namely as being uniquely identified by their source and target nodes (recall the graph model of relational growth grammars (Sect. 5.2.2 on page 97) and the variant of the base implementation (Sect. 9.1.1 on page 235), which is also used for GroIMP).

Therefore, we developed an own implementation of the required features in the package `de.grogra.persistence`, but with several ideas adopted from JDO. For example, there is a distinction between *first-class objects* and *second-class objects*: first-class objects have an identity on their own, represented by a unique identifier of type **long**, while second-class objects do not have such an identity and are only addressable as attribute values of first-class objects (or, and this extends JDO, indirectly as components of attribute values, see the above requirement of fine-grained access). First-class objects have to implement `PersistenceCapable` and are managed, including their life cycle, by a `PersistenceManager`. In the only currently existing usage of this system, namely the graph of GroIMP, first-class objects are exactly the nodes.

As an exception to the rule that second-class objects do not have an identity we added the interface `Shareable`. Shareable objects can be shared among several first- or second-class objects, and this information is used when reading from and writing to a data stream like the persistence storage in a file or the change log. As an example, consider the list of predefined shaders which GroIMP provides. These are second-class objects, but as they implement `Shareable`, they can be read from and written to streams as references to GroIMP. Otherwise, their origin would get lost when re-reading, i. e., we would have a new shader for each occurrence which is equal to the one provided by GroIMP, but not the same.

The implementation supports reading from and writing to both binary and XML-based formats. The binary format is intended for the exchange of data and transactions between connected graph instances, while the XML format is intended for long-term storage in files.

A.3.3 Graph Implementation

The graph implementation in the package `de.grogra.graph.impl` is defined on top of the previously described packages. It implements the graph interface and uses the persistence system to handle modifications, change logs and persistent storage. The class diagram is shown in Fig. A.4 on the next page.

Internally, the topology is stored by doubly-linked lists of incident edges for each node. The required two links are stored within an `Edge` itself. A speciality is that `Node` is a subclass of the abstract class `Edge`. This is an optimization with respect to memory usage: given that RGG structures can quickly become large, but usually are tree-like for the most part, we have about n edges for n nodes, and most nodes have exactly one incoming edge. Now in our setting, a `Node` can at the same time represent the information of a single incoming edge. Thus, only for nodes with more than one incoming edge there is the need to actually allocate memory for edge objects from the heap (which are then of class `EdgeImpl`). As each Java object introduces a memory overhead and also makes the task of the garbage collector more difficult, saving n objects is clearly advantageous.

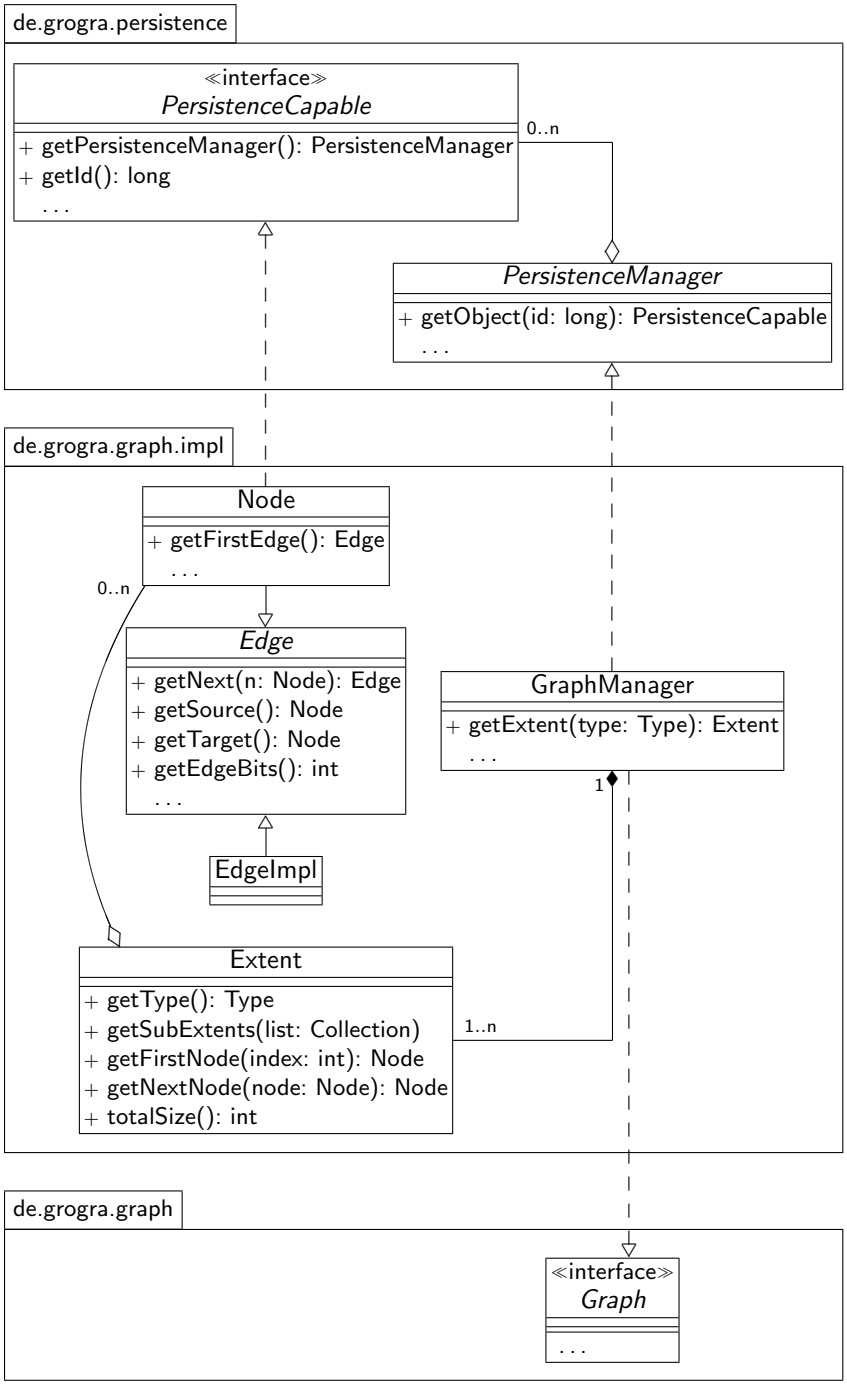


Figure A.4. Class diagram for graph implementation

As required by the interface `Graph`, a `GraphManager` maintains a set of root nodes, identified by a `String`. Only those nodes which can be reached from at least one of the root nodes by a path of edges are kept in the graph. In other words, the graph of a `GraphManager` consists of connected components each of which has to contain at least one root node. As a consequence, the deletion of a node or edge in a tree graph leads to the deletion of its whole subtree.

A special feature of the `GraphManager` is the management of extents. For each used node class in the graph, there exists an `Extent` which collects all nodes of this class and also has a link to the extent of its superclass and to the extents of its subclasses. This feature is currently only used within the RGG plug-in for a fast processing of queries which specify node type patterns like in the rule `F(x) ==> F(2*x);`. Instead of having to scan the whole graph for nodes of class `F`, the corresponding extent is used together with its subextents to directly iterate over all nodes of class `F`. The method `totalSize` could be used for a graph statistics to optimize the search plan of a query (see Sect. 7.3.1 on page 197), but this is not yet done.

An extent collects its nodes not by a single list, but by a number of (doubly linked) lists. Currently, eight lists are used, but this can easily be changed. This feature is used to partition the extents. An example of its usage is the specification of a model which creates a large graph, but where the rules are only applied to (possibly varying) parts of the graph by specifying which extent lists shall be considered for queries. This technique was used for the alder model of Sect. 10.5.9 on page 332 where the main stem and branches were constructed one after another. Another application is to prevent structures from being transformed by rules.

The persistence mechanism of the previous section requires some additional structures for node classes so that it is able to manage nodes and their field values. Among these structures is, within each node class, the declaration and initialization of a special static field `$TYPE` and the definition of the method `getNTypeImpl` to return this `$TYPE` and of the factory method `newInstance` to return a new instance of the class. To automatize the generation of these structures, the source distribution of GroIMP contains a source code preprocessor which scans source files for special preprocessing instructions and then creates the required code automatically.

As structures created by relational growth grammars may be very deep (e. g., the snowflake example of Sect. 10.1.1 on page 269 creates a linear chain of $6 \cdot 4^n - 1$ nodes after n steps), the implementation of recursive graph algorithms by recursive method invocations would quickly result in a stack overflow. Therefore, recursive method invocations have to be avoided by using an explicit stack-like data structure in implementations. This has been done, e. g., in the implementation of the `accept` method (Sect. A.3.1 on page 379) which provides a hierarchical visitor with the graph structure.

A.4 Projects

The current state of the graph and the GroIMP workbench including lists of interactively defined auxiliary objects (e. g., shaders), source files, images and the layout of the graphical user interface constitutes a *project* of GroIMP. All state information can be saved to and retrieved from a hierarchy of files, where source files, images and other data intrinsically represented by a file are stored as they are, while the graph and the state of the workbench are stored as XML files. This hierarchy of files may either be written directly to the file system, in which case the file name extension of the main file is `gs`, or it may be written to a single ZIP archive containing all files, in which case the file name extension is `gsz`. The ZIP archive follows the format of JAR files [183], i. e., its first entry is the special file `META-INF/MANIFEST.MF` which describes the MIME-type and possible further information of the other files. Using a format based on XML and ZIP is not a new idea, this is also used by the OASIS Open Office XML Format as it is advantageous for several reasons [187]:

- XML is human readable and is thus accessible even without the corresponding application.
- There exist a lot of standards and tools how to process XML files.
- ZIP compression yields a single small file and can handle both binary and textual files.
- The content of ZIP files can be looked at and modified by standard ZIP tools.

As an example, the content of the file `Ludo.gsz` of the Ludo model (Sect. 10.8.1 on page 347) is as follows:

```
Ludo.gsz
├─ META-INF
│  └─ MANIFEST.MF
├─ data
│  └─ Pawn.con ..... defines profile of surface of revolution for pawns
├─ images
│  └─ Die.png ..... texture image for die
├─ project.gs ..... main project file
├─ graph.xml ..... graph representation
└─ Ludo.rgg ..... source code
```

A.5 Graphical User Interface

The graphical user interface of a GroIMP project consists of a set of panels. There are panels for the source code editor, for the 3D view, for images, shaders, toolbars and so on. Usually, only a subset of the available panels is

shown at a time. This subset and its layout (i. e., the arrangement of the panels) is at the discretion of the user: any not shown panel can be opened via the menu Panels, the user may change the layout by a Drag&Drop-mechanism for the panels. There also exists a list of predefined layouts in the menu Panels/Set Layout, and the user may also add the current layout to this list.

An important subset of the panels consists of explorer panels which show a list of objects of specific types. Most of them are available under the menu Panels/Explorers. E. g., the image explorer shows all images of the current project, the shader explorer all shaders, the object explorer all "stand-alone" objects which in fact are nodes (or even whole graphs, referenced by an anchor node). Objects may be added to such a list by choosing a submenu of the menu Objects/New within the explorer panel. They may also be deleted by the key Delete or renamed by clicking twice on the object or by the key F2.

The file explorer manages a list of "raw" files. They can be source code, JAR libraries, or HTML or plain text files with some information about the project.

A.6 Import and Export Filters

GroIMP contains a lot of import and export filters. Table A.1 summarizes the most important filters contained in version 0.9.8.1 of GroIMP. Most import filters are used within the menu Objects/New/From File of an explorer panel to create a new object of the specific type out of a description in a file. These can then be referenced in the user interface where required, e. g., an image can be referenced as texture in a shader, or they can also be referenced in the source code of an RGG model, see Sect. B.13.5 on page 411.

A lot of import filters like GraphML, DXF or CPFG surfaces read in a whole graph, referenced by an anchor node. Besides adding such a graph to the object explorer, it can also be added directly to the scene graph of GroIMP by the menu Objects/Insert File of the main window.

A.6.1 GraphML Import

The GraphML import filter reads a description of a graph conforming to the GraphML format [14]. It defines five GraphML attributes: `type` and `value` exist for both nodes and edges, where `type` specifies the class to use for the object and `value` is a whitespace-separated list of assignments of values to the instance variables. If the `type`-attribute is used for an edge, it is assumed to actually refer to a node class, and a compound edge is created from the node (see Sect. A.3.1 on page 382). Alternatively, one can use the attribute `bits` to specify the edge bits of a normal edge as a comma-separated list of edge types (see the API documentation for the exact format). The following is an example of a scene consisting of a cylinder bearing a cone by a branch edge (symbol `+`):

Format	Extension	Import/Export	Plug-in	Explanation
Source code	java, xl, rgg	import	RGG	Sect. B.1
GROGRA source code	lsy, ssy	import	RGG	Sect. B.15
JAR library	jar	import	Platform	Sect. B.1
GraphML	graphml	import	Platform	Sect. A.6.1
X3D	x3d	both	X3D	
POV-Ray	pov	export	POV-Ray	
AutoCAD DXF	dxf	both	DXF	
Wavefront Object	obj	import	DXF	
GROGRA DTD, DTG	dtd, dtg	import	RGG	Sect. A.6.2
MTG	mtg	import	Platform	[172]
MSML	msml	both	Platform	[172]
CPFG function	func	import	CPFG	
CPFG surface	s	import	CPFG	
CPFG contour	con	import	CPFG	
Xfrog	xfr	import	3D-CS	[77]
Protein Data Bank	pdb	import	PDB	
Image: any format supported by the ImageIO API of the JRE, additionally import from portable pixel maps and HGT height fields and export to OpenEXR HDR images	png, jpg, gif, ... ppm, hgt, exr	both	IMP	
Encapsulated Postscript	eps	export	IMP	

Table A.1. Supported data formats of GroIMP, version 0.9.8.1

```
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
  <key id="ntype" attr.name="type" for="node"/>
  <key id="nval" attr.name="value" for="node"/>
  <key id="etype" attr.name="type" for="edge"/>
  <key id="eval" attr.name="value" for="edge"/>
  <key id="bits" attr.name="bits" for="edge"/>
  <node id="s">
    <data key="ntype">de.grogra.imp3d.objects.Cylinder</data>
    <data key="nval">radius=0.2 length=2</data>
  </node>
  <node id="c">
    <data key="ntype">de.grogra.imp3d.objects.Sphere</data>
  </node>
  <edge source="s" target="c">
    <data key="bits">+</data>
  </edge>
</graphml>
```

A.6.2 DTD and DTG Import

DTD files ("descriptive tree data format") typically contain the results of geometric and topological measurements of plants [108, 106]. The format is one of the input formats of the GROGRA software (Sect. 3.15.1 on page 35). The DTG format is the native data format of the GROGRA software and completely represents the current 3D structure.

Both formats can be imported to GroIMP and return a tree of nodes of class `de.grogra.grogra.DTGShoot`, referenced by its root. A possible usage is the analysis of tree data by XL queries, see Sect. 10.5.5 on page 314.

A.7 3D Plug-In

The plug-in IMP-3D provides 3D functionality and, thus, is one of the most important plug-ins of GroIMP. The plug-in uses a graph given by the interface of Sect. A.3.1 on page 379 and interprets this as a 3D scene graph [59, 82, 84] in several contexts, namely when rendering it both in real-time (AWT-based wireframe display, OpenGL-based display with surface shading) and by ray-tracing, when exporting it to 3D data formats, and when doing geometric computations in RGG models (e. g., computation of distance, intersection tests, radiation model). A speciality of the scene graph is that each object may specify two coordinate transformations: the first one is applied to the inherited coordinate system of the parent and defines the local coordinate system of the node itself, the second one is applied to this coordinate system and defines the coordinate system which is passed to children. An application thereof is found in the classes having an axis such as `Cylinder` or `Cone`: these may be oriented in an arbitrary way with respect to their parent, and the coordinate system passed to children is shifted along the axis such that its origin coincides with the top of the axis. As a consequence, in a sequence of nodes of these types each base point of a successor coincides with the tip of its predecessor, which exactly conforms with the turtle interpretation of the F command. (In fact, this is only the default behaviour, but the classes implement a more general mechanism where the relative position along the axis can be specified.)

From an RGG modeller's perspective, two packages are of great importance: `de.grogra.imp3d.objects` and `de.grogra.imp3d.shading`. The first one defines a lot of classes to be used for nodes in a 3D scene graph. The hierarchy is shown in Fig. A.5 on the facing page. Its base class `Null` is nothing but an invisible `Node` with a transformation attribute, i. e., it transforms the local coordinate system of its children. The subclass `ShadedNull` has an additional attribute for a shader which is used within the subclasses to shade their surfaces. The subclasses of `ColoredNull` do not have a surface as they are locally 0- or 1-dimensional, therefore a simple colour attribute is sufficient

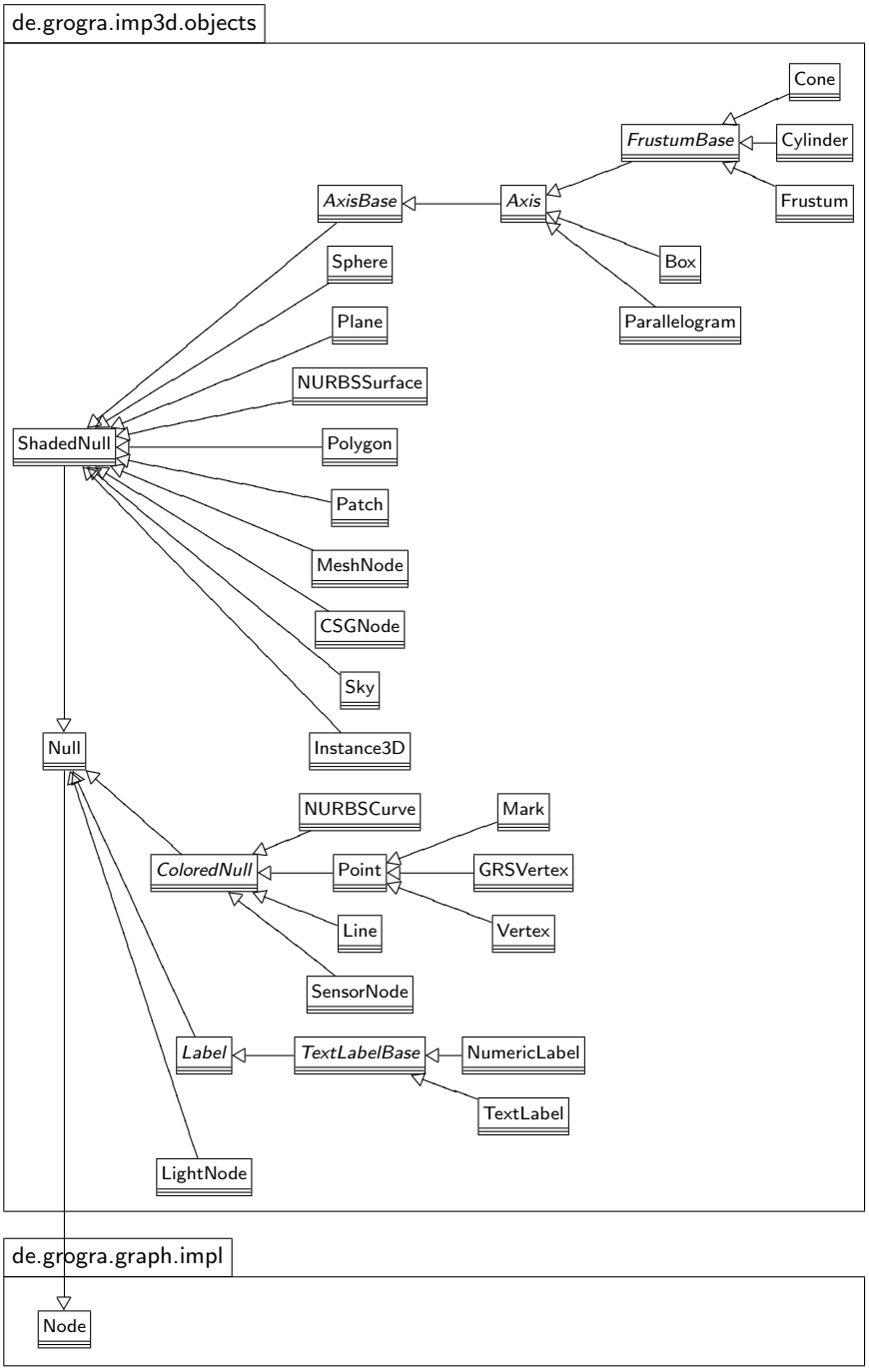


Figure A.5. Class diagram for 3D scene graph nodes

for their visual representation. The illumination of a scene is done by light sources which can be inserted either as part of a `LightNode` or of a `Sky`.

Besides geometric primitives like `Sphere`, `Box`, `Line`, there is also more complex 3D geometry. `NURBSSurface` and `NURBSCurve` have an attribute of type `de.grogra.math.BSplineSurface` or `de.grogra.math.BSplineCurve`, respectively, and render the surface or curve in three dimensions, using the framework of non-uniform rational B-splines (NURBS, [149]). GroIMP provides a lot of construction techniques for such surfaces and curves. `Polygon` takes a `de.grogra.math.VertexList` as input and draws a single polygon whose boundary passes the vertices in the list. `Patch` has a rectangular `de.grogra.math.VertexGrid` as input and draws a corresponding polygon mesh. Finally, `MeshNode` is used as the 3D scene graph node for arbitrary polygon meshes (e. g., specified by a graph rotation system, Sect. 10.6 on page 335).

The other important package `de.grogra.imp3d.shading` does not define 3D scene graph nodes, but is used for optical properties like shader or light source attributes. Figure A.6 on the facing page shows the hierarchy of its most useful classes. `RGBAShader` is a simple Lambertian shader specified by an RGB colour and an α-value for transparency. `Phong` is based on the Phong model and is controlled by several properties. For most of these properties, textures can be used which are either procedural or based on an image. `SunSkyLight` can be used as a sky background, but it also implements a light source which realistically models the light distribution of the true sun and sky. The two `SwitchShader` subclasses choose the shader to be used from a list of shaders: `SideSwitchShader` chooses among two shaders, one for the front side of a surface, the other for the back side. `AlgorithmSwitchShader` chooses among three shaders, one for the visualization in the user interface, one for the use in ray tracing, one for the radiation model. The latter then has to be physically correct in order to obtain meaningful results from the radiation model, while the other ones can be chosen to allow a nice visualization.

The subclasses of `LightBase` implement different kinds of light sources. These are used as light attribute of a scene graph node of class `LightNode`. The 3D scene graph node `Parallelogram` can not only specify geometry, but may also be an area light source.

A.7.1 Built-In Raytracer

The built-in raytracer Twilight is actually implemented in the plug-in Raytracer. The plug-in IMP-3D implements the geometry and shading interfaces of the raytracer so that the latter can be used to obtain high-quality renderings of the 3D scene graph. In fact, most renderings of Chap. 10 were rendered by Twilight. Figure A.7 on page 394 shows a rendering of a simple scene with diffuse reflection, an area light and a lens projection with noticeable depth of field. All these effects are simulated by the path tracer option of Twilight,

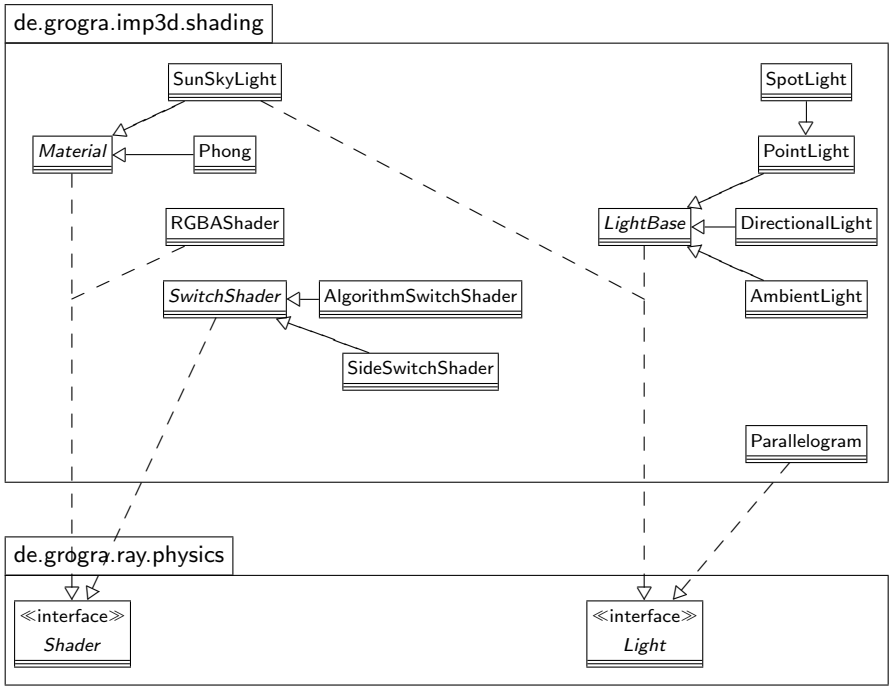


Figure A.6. Class diagram for shaders and light sources

i. e., a Monte Carlo-based approximation of the rendering equation for global illumination [89, 195].

For an efficient Monte Carlo algorithm, the shaders implemented by the plug-in IMP-3D (`RGBAShader`, `Phong`, `SunSkyLight`, see Fig. A.6) create pseudorandom samples based on importance sampling [195], i. e., samples whose contribution is likely to be more important are chosen with a higher probability than samples which are relatively unimportant.

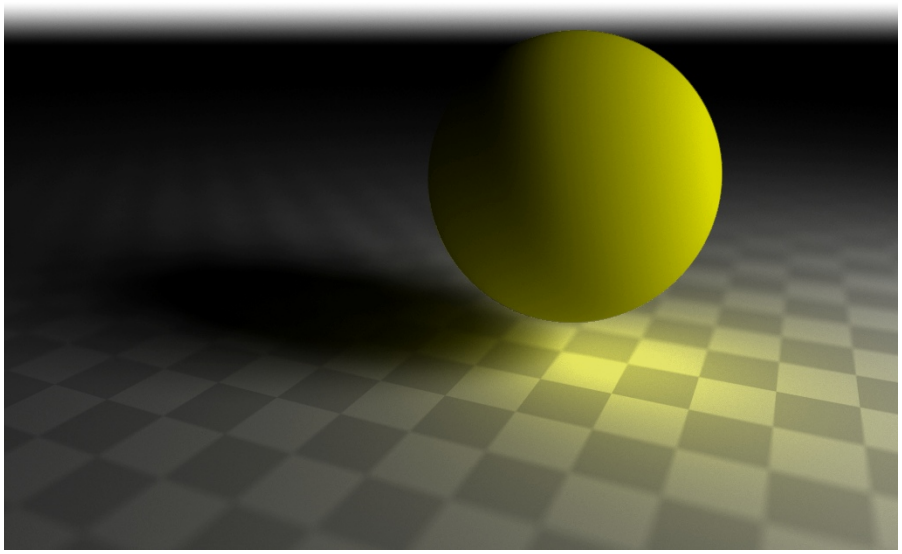


Figure A.7. A simple scene rendered by the path tracer option of Twilight. An area light illuminates the scene from the right. The sphere casts a deepest shadow, enclosed by a soft shadow due to the extent of the area light. The diffuse reflection of yellow light can be seen on the ground under the sphere, it is also responsible for the illumination of the sphere in the shadow of direct light. These effects are amplified in the image for illustration purposes. The blurred chessboard pattern shows the effect of depth of field.

B

The RGG Plug-In of GroIMP

The RGG plug-in establishes the link between GroIMP and the XL programming language. It uses the base implementation of the XL interfaces (Sect. 9.1 on page 235) to make the graph of GroIMP (Sect. A.3.3 on page 384) accessible to the run-time system of the XL programming language and to provide a derivation mechanism which is compliant with relational growth grammars. This plug-in also provides a re-implementation of some of the features of the GROGRA software within the new framework.

B.1 Overview of Functionality

The RGG plug-in is dedicated to modelling by (rule-based) programming, so its first task is to compile source code into executable code. For this purpose, any source code file listed in the file explorer (extensions `java`, `xl`, `rgg`, `lsy`, `ssy`) is considered to be a part of the model, and the XL compiler is invoked to compile all source code files simultaneously. Furthermore, if there are JAR libraries listed in the file explorer, these are implicitly added to the classpath of source code.

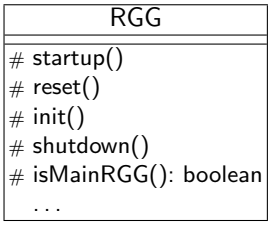
To facilitate the execution of models, the plug-in handles compiled classes which extend `de.grogra.rgg.RGG` specially (Sect. B.2 on the next page). There are dedicated methods for the life cycle of such RGGs, and public methods without argument can be invoked directly by the RGG toolbar as they are considered to expose the basic actions of the model.

Finally, the plug-in provides a large collection of useful classes and methods. A lot of scene graph nodes with the semantics of turtle commands are defined in the package `de.grogra.turtle` following the naming convention of the GROGRA software, see Table B.2. The class `Library` defines utility functions related to the topology and 3D geometry of the scene graph, to the control of rule application and to some further tasks (Sect. B.13 on page 408). The radiation model is also contained in this plug-in and described later in Sect. B.14 on page 412.

An easy starting point for RGG modelling within GroIMP is via the menu File/New/RGG Project. The user is asked for the name of the main class, and then a new project with a simple model is opened. This automatically uses the RGG layout of panels (Sect. A.5 on page 387) which contains, among others, the 3D view, source code editor and the RGG toolbar. But this layout can also be obtained for any project by choosing it via the menu Panels/Set Layout/RGG Layout.

B.2 `RGG` Class and Its Life Cycle

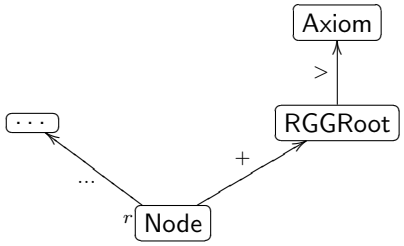
The class `de.grogra.rgg.RGG` stands for a relational growth grammar. Whenever a compiled class is a subclass of `RGG`, a singleton instance of the class is automatically created and added to the list of meta objects (accessible through the meta objects explorer). The following methods are used to notify the instance about its life cycle:



The method `startup` is invoked when an `RGG` instance is created within GroIMP, i. e., after the re-compilation of its source code and after loading a project already containing the instance. Afterwards, `reset` is invoked if the instance has been created due to a re-compilation of source code. It is not invoked if the instance was loaded from a project as in this case it is assumed that the read project data already specifies a state where the method `reset` has been invoked.

Unless `isMainRGG` returns **false**, the default implementation of `startup` scans the used subclass of `RGG` for public non-static methods without parameters and having return type **void**. Each such method is considered to be an action of the model (e. g., a collection of rules which constitute the dynamics of a single transformation step), and a button to invoke the method is added to the RGG toolbar.

Again unless `isMainRGG` returns **false**, the default implementation of `reset` establishes an initial structure in the graph, comparable to a fixed start word in the context of L-systems:



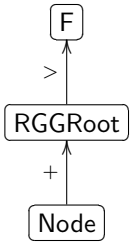
If there already exists an RGGRoot node connected to the main root r by a branch edge, reset removes this node together with its subtree. Like RGG, the classes RGGRoot and Axiom are members of the package de.grogra.rgg. After creating the start graph, the default implementation of reset invokes init. The default implementation of the latter does nothing, a typical implementation in subclasses replaces the Axiom by some model-specific initial structure.

The last life cycle method shutdown is invoked on the RGG instance when it is unloaded, i. e., when a new version of its source code is compiled and the old instance is removed.

For the example

```
import de.grogra.rgg.*;
import de.grogra.imp3d.objects.*;

public class Model extends RGG {
    protected void init() [
        Axiom ==> Box;
    ]

    public void step() [
        Box ==> Cone Box;
    ]
}
```

the structure after complete initialization is



The RGG toolbar shows a button labelled step which invokes the corresponding method when clicked. There always is a Reset button to invoke the reset method which then re-establishes the structure after initialization. As described above, this only removes the subgraph starting at the single RGGRoot,

but does not affect nodes directly connected with the main root of the graph (and their descendants). In other words, there is a partition of the graph in a part which is considered to be algorithmically generated (the part reachable via `RGGRoot`) and is thus removed on reset, and a part which is considered to be created by other means (e. g., interactively like the terrain, sky and pavilion in the stand model of Sect. 10.5.6 on page 326) and is thus not removed on reset.

The default implementation of the method `isMainRGG` returns **true**. However, if there is more than one subclass of `RGG` in a project, one has to override this method in the subclasses so that only one returns **true**. The corresponding class is then used for initialization and resetting of the model, while the others behave almost like ordinary classes, i. e., an instance of them is automatically created, but the methods `startup` and `reset` do nothing, and no menu entries for methods are created.

B.3 XL Console

The XL console is a panel of the user interface. It is used for two purposes: some functions of the library (Sect. B.13 on page 408) write textual output to the console, and the user may type in XL statements which are compiled and executed immediately. A valid statement within the console is any valid statement of the XL programming language, but the terminal semicolon may be omitted. Variables are implicitly declared by assignments and remembered throughout the same session of the console. The result of non-void statements is written to the console. The automatic imports of the RGG dialect (see next section) are also automatic for the console. Further import statements may be given, these are then used throughout the same session. An example session is

```
> count((*F*))
3
> (*F*)[length]
11.0
10.0
9.0
> sum((*F*)[length])
30.0
> x = mean((*F*)[length])
10.0
> (*F*)[length] = x
10.0
10.0
10.0
>
```

B.4 RGG Dialect of the XL Programming Language

The XL programming language being a true extension of the Java programming language, there is some syntactical overhead when specifying RGG models. For example, a possible implementation of the snowflake construction looks like:

```
import de.grogra.rgg.*; // for Axiom, RGG
import de.grogra.turtle.*; // for F, RU

public class Koch extends RGG {
    public void rules() [
        Axiom ==> F(10) RU(120) F(10) RU(120) F(10);
        F(x) ==> F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3);
    ]
}
```

From the perspective of a user familiar with L-systems, one would ideally have to specify only the two lines for the actual rules. To facilitate RGG modelling, we defined a simplified dialect of the XL programming language called the *RGG dialect* which is indicated by a source file having the extension `rgg`. This dialect implicitly surrounds the source code by a declaration of a subclass of `RGG`, and it adds several automatic imports. For a source file `Model.rgg`, the equivalent XL code is

```
import de.grogra.xl.lang.*;
import de.grogra.annotation.*;
import de.grogra.rgg.*;
import de.grogra.turtle.*;
import de.grogra.imp3d.objects.*;
import de.grogra.imp3d.shading.*;
import de.grogra.math.*;
import javax.vecmath.*;
import de.grogra.pf.data.*;

import de.grogra.graph.impl.Node;
import de.grogra.rgg.Attributes;

import static de.grogra.xl.util.Operators.*;
import static de.grogra.rgg.Library.*;
import static de.grogra.vecmath.VecmathOperators.*;
import static de.grogra.imp3d.shading.RGBAShader.*;
import static de.grogra.pf.ui.ChartPanel.*;
import static de.grogra.turtle.TurtleState.*;
import static de.grogra.xl.impl.base.Graph.*;

imports declared in Model.rgg

@ImplicitDoubleToFloat(true)
```

```
public class Model extends RGG {
    public static Model INSTANCE;

    {INSTANCE = this;}

    declarations of Model.rgg, member classes are implicitly static
}
```

Thus, the imports most frequently needed are automatic, and the singleton nature of `RGG` subclasses is reflected by the static field `INSTANCE` by which the single instance can be accessed. Furthermore, within the RGG dialect there is no need to cast **double** values to **float** as this is an implicit conversion.

B.5 Implicit Annotations for Source Code

When the XL compiler is invoked within the RGG plug-in, the following annotations are implicitly added to compilation units, i. e., to each source file:

```
@de.grogra.xl.query.UseModel(Compiletime.class)
@de.grogra.xl.property.UseModel(type=Node.class,
                                model=PropertyCompiletime.class)
@de.grogra.xl.modules.DefaultModuleSuperclass(Node.class)
@de.grogra.xl.modules.InstantiationProducerType(Instantiation.class)
@de.grogra.xl.compiler.UseExtension(CompilerExtension.class)
```

The RGG-specific classes used in the above annotations are defined in the package `de.grogra.rgg.model`. The last annotation defines a compiler extension which is described in the next section.

B.6 Processing of Compiled Classes

The implicit annotation `@UseExtension(CompilerExtension.class)` makes use of the possibility to define extensions for the XL compiler (Sect. 8.6 on page 226). The used extension `de.grogra.rgg.model.CompilerExtension` processes compiled classes by several steps:

- Types are made public.
- If a node class declares a constructor with no arguments, it is made public. Otherwise, an implicit public constructor with no arguments is created.
- Some additional structure is added which is required by the persistence mechanism of Sect. A.3.2 on page 383. This includes the declaration and initialization of a special static field `$TYPE`. This additional structure has to be created also for the predefined node classes of GroIMP, but as these are implemented in the Java programming language, we cannot make use of a comparable compiler extension, but have to specify the additional structure in the source code itself (see Sect. A.3.3 on page 384).

- For a module declaration with instantiation rule (Sect. 6.11.2 on page 182), the method `getInstantiator` of class `Node` is overridden so that the instancing mechanism of the graph interface (Sect. A.3.1 on page 382) uses the instantiation rule.

Making node classes and their default constructors public is required by the persistence mechanism (otherwise it would not be able to create instances of node classes when reading their data from a stream). Making any type public is just for convenience, as we are only then allowed to access them in the XL console. But note that the change of accessibility to public is done as a postprocessing step so that this is only reflected in the bytecode and then in the classes at run-time, but the original accessibility is used throughout the proper compilation.

B.7 Implementation of Graph Model

The XL graph interface is implemented by `de.grogra.rgg.model.RGGGraph`. The implementation of the method `enumerateNodes`, whose task is to enumerate all nodes of the graph having a given type, uses the mechanism of extents of the graph of GroIMP to efficiently find all suitable nodes (Sect. A.3.3 on page 386). This mechanism partitions each extent in a set of numbered lists, and by invoking the method `setVisibleExtents`, one may choose those lists which shall be considered by `enumerateNodes`. By default, all but the last list are visible. I. e., by moving a node to the last list one may screen the node from further rule applications. The extent list of a node may be changed either by directly setting the property `extentIndex` of the node, or by invoking utility methods in the `Library` (see Sect. B.13 on page 408).

The implementation of the method `enumerateEdges`, whose task is to enumerate all edges of a given node fulfilling some conditions, is able to handle normal edges of the graph of GroIMP (based on the encoding of types in an **int**-value), but also compound edges with an interjacent node playing the role of the edge information. Therefore, we may not only write a pattern like `A -successor-> B` to find all nodes of type `A` with a successor edge to a node of type `B` (`successor` is a constant defined in the `Library`), but also a pattern like `A -E-> B` to look for compound edges with node type `E`, i. e., for actual edges of type `EDGENODE_IN_EDGE` from the `A`-node to the `E`-node and of type `EDGENODE_OUT_EDGE` from the `E`-node to the `B`-node (Sect. A.3.1 on page 382).

B.8 Operations of the Producer

The producer type for right-hand sides is specified by the query model of the RGG plug-in and is `de.grogra.rgg.model.RGGProducer`. It extends the producer of the base implementation (Sect. 9.1.4 on page 243) and implements

the prefix operators for node expression shown in Table B.1. Furthermore, the special producer method `producer$getRoot` (the internal meaning of the root symbol `^`, Sect. 6.7.2 on page 165) is implemented and returns the root node of the RGG-generated part of the graph, i. e., the instance of `RGGRoot` as described in Sect. B.2 on page 396. The three methods `producer$push`, `producer$pop` and `producer$separate`, which are responsible for the implementation of the symbols `[`, `]` and `,` (Sect. 6.7.4 on page 168), are also implemented using the corresponding helper methods of the base implementation. Some examples of the usage of the producer operations are

```
A [B] C
A -(branch|successor)-> B
A [ > B] -E-> C
```

Assuming that all uppercase letters stand for node classes, the first example defines a graph with a branch edge from the `A`-node to the `B`-node and a successor edge from the `A`-node to the `C`-node. The second example uses the predefined constants `branch`, `successor` of the `Library` class and defines a graph with both a successor and a branch edge from the `A`-node to the `B`-node. Like the first example, the third example creates edges from the `A`-node to both the `B`- and `C`-node, but the default branch edge of bracketed parts is overridden by the operator `>` to be a successor edge, and the edge from the `A`-node to the `C`-node is in fact a compound edge with an interjacent `E`-node, an edge of type `EDGENODE_IN_EDGE` from the `A`-node to the `E`-node and an edge of type `EDGENODE_OUT_EDGE` from the `E`-node to the `B`-node (Sect. A.3.1 on page 382).

To facilitate the usage of the vertex-vertex algebra implementation, the `RGGProducer` also declares a method `vv` which returns a `VVProducer`, and an overloading for the whitespace operator with a `VVProducer` parameter which returns the passed argument. For an example, see Sect. 10.6 on page 335.

B.9 Properties

The implementation of properties is based on the persistence mechanism (Sect. A.3.2 on page 383): for any field of a class which is registered with this mechanism there is also a property according to Sect. 6.10 on page 174. E. g., the class `de.grogra.imp3d.objects.Parallelogram` has the field `axis` of type `Vector3f`. This is also available as property `axis` with three subproperties `x`, `y`, `z`. For any property type, all meaningful operators for deferred assignments are implemented (namely, the methods of the interfaces shown in Fig. 9.5 on page 252). They append corresponding entries to a property modification queue as suggested in Sect. 9.1.8 on page 251. Thus, deferred assignments take effect when modification queues are processed, i. e., when the current parallel derivation is applied (Sect. 9.1.2 on page 237).

Properties are only defined with respect to first-class objects, i. e., nodes (Sect. A.3.2 on page 383). This means that in a property variable `x[property]`

Prefix Operator	Description
whitespace	add node and connect with previous node, if any, by a successor edge
`>`	connect node with previous by a successor edge
`<`	connect node with previous by a reverse successor edge
`<->`	connect node with previous by successor edges in both directions
`---`	connect node with previous by a successor edge if there does not yet exist such an edge in either direction
`+>`, `<+`, `<+>`, `-+-`	as before, but branch instead of successor edges
`/>`, `</`, `</>`, `-/-`	as before, but refinement instead of successor edges
`-`*e*`->`, `<-`*e*`-`, `<-`*e*`->`, `-`*e*`-` with an **int**-valued expression *e*	as before, but edges of type *e* instead of successor edges
`-`*n*`->` with a node-valued expression *n*	create a compound edge with *n* encoding the edge data (for compound edges see Sect. A.3.1 on page 382)
`<-`*n*`-` with a node-valued expression *n*	as before, but reverse direction

Table B.1. Prefix operators for node expressions provided by `RGGProducer`

the instance `x` has to be a node. Therefore, given an instance `v` of type `Vector3f`, there is no property variable `v[z]`, although we have, e. g., a property variable `p[axis][z]` for a `Parallelogram` `p`. In other words, second-class objects may only declare subproperties of properties of first-class objects.

The difference between normal instance variables like `p.axis` or `p.axis.z` and property variables `p[axis]` or `p[axis][z]` is not only that the latter may be used as targets of deferred assignments. When new values are assigned to these variables, an assignment to a normal instance variable like `p.axis.z = 1;` does not trigger any further action. Contrary, an assignment to a property variable `p[axis][z] = 1;` is performed through the persistence mechanism, and this offers a lot of useful side effects to be executed on such assignment. By default, all changes are logged to a protocol (Sect. A.3.2 on page 383) so that they may be undone or transferred over a network to other instances of the same graph which shall be kept consistent with each other. Furthermore, listeners may be added to the graph which then are notified about changes like the assignment of new values to variables. This is used by GroIMP to automatically trigger a redraw of graph views like the 3D view. Therefore, the following rule should be obeyed: attributes of nodes which are already part of the graph must not be modified by assignments to their conventional instance variables, but only by assignments to their property variables. As setter-methods like `setAxis` only wrap assignments to instance variables, these also must not be used for nodes which are already part of the graph. However, it is perfectly legal to use conventional assignments or setter methods on newly created nodes which are not yet part of the graph as in

```
Axiom ==> Parallelogram.(setLength(4), setAxis(1, 0, 1));
```

A later enlargement of the parallelogram should then be implemented using property variables:

```
p:Parallelogram ::> {p[length] :*= 1.1; p[axis][x] :*= 1.1;}
```

B.10 Wrappers

The XL programming language allows to include non-node types in queries by suitable wrapper nodes (Sect. 6.5.2 on page 148). The RGG plug-in defines the node classes `BooleanNode`, ..., `ObjectNode` which wrap arbitrary primitive values or objects in their property `value`, and these are integrated into the query mechanism by the compile-time model so that one can use a pattern `x:String` which finds all wrapped strings `x` or even `x:"FSPM"` to find all strings equal to `"FSPM"` (together with their wrapper nodes `$x`). Furthermore, the two 3D node classes `NURBSCurve` and `NURBSSurface` are also considered as wrapper classes, their wrapped values being of type `BSplineCurve` or `BSplineSurface`, respectively. Therefore, a pattern like `c:Circle` is valid although a `Circle` is not a node by itself. Likewise, we may write a rule

```
c:Circle ::> c[radius] :*= 1.1;
```

By using the mechanism discussed in Sect. 6.10.4 on page 179, this is actually interpreted as

```
b:NURBSCurve & (b[curve] instanceof Circle) ::>
    ((Circle) b[curve])[radius] :*= 1.1;
```

since properties are only defined with respect to node classes, but not with respect to second-class objects (see the previous section).

While the compile-time and run-time interfaces have to deal with how to unwrap values out of wrapper nodes, the wrapping of values in wrapper nodes is not covered. As this typically happens on right-hand sides, the first candidate where to specify wrapping is within the implementations of prefix operators in the producer. A more convenient way is to use user-defined conversions from wrappable values to their wrapper nodes, these conversions are then automatically used when the producer does not define a suitable operator method. `RGGProducer` defines such conversions from any wrappable value as static conversion methods. By the general mechanism of producers explained in Sect. 6.7.1 on page 164, these are in scope on right-hand sides and are thus implicitly used for conversion when required.

For the unwrapping of values there exist user-defined conversions in the `Library`, see Sect. B.13.7 on page 412.

B.11 Interpretive Mode

The base implementation already provides a general framework for interpretive rules. It is explained in detail in Sect. 9.1.6 on page 248. There, the type IM for interpretive marks was used; within the RGG plug-in the concrete class `de.grogra.rgg.model.InterpretiveMark` is used for this purpose. The class `RGGGraph` defines the two methods `removeInterpretiveNodes` and `removeInterpretiveNodesOnDerivation` to coordinate the deletion of interpretive structures. The first one immediately deletes these structures by looking for all interpretive marks, finding the connector nodes of the interpretive structure indicated by the mark, and then removing the whole interpretive structure as discussed in Sect. 9.1.6. The latter method just sets a flag that the first method shall be invoked on derivation, namely as first action of the `derive` method of `RGGGraph`. In effect, this adds a rule for deletion of interpretive structures to the set of applied (generative) rules.

Using the method `rggGraph` of the `Library` which returns the currently used `RGGGraph`, a possible usage of interpretive rules is the following:

```
rggGraph().setDerivationMode(PARALLEL_MODE);
rggGraph().removeInterpretiveNodesOnDerivation();
[
    A ==> [B] X A;
]
derive();
rggGraph().setDerivationMode(PARALLEL_MODE | INTERPRETIVE_FLAG);
[
    X ==> RH(180) F(1);
    B ==> RU(60) F(1);
]
derive();
```

For the first rule block, we use the normal parallel derivation mode (see Sect. 9.1.5 on page 245), but remove any existing interpretive structures on derivation. After applying the resulting parallel derivation, we switch to the interpretive mode and assign a geometric meaning to internode-like nodes of class `X` and branches of class `B`.

To facilitate the usage of interpretive rules, the `RGG` class defines the helper method `applyInterpretation` which basically implements the code from above and delegates to the method `interpret` to invoke the actual interpretive rules. This method is empty in `RGG`, but may be overridden in subclasses as in this example:

```
public void step() [
    A ==> [B] X A;
    {applyInterpretation();}
]

protected void interpret() [
```

```
    X ==> RH(180) F(1);
    B ==> RU(60) F(1);
]
```

Note that the instantiation rules presented in Sect. 6.11.2 on page 182 provide a similar framework to equip nodes with a geometric meaning. As it has been discussed there, both frameworks have advantages and disadvantages, but a general advice is to prefer instantiation rules instead of interpretive rules, especially for large graphs where the efficiency concerning space and time is crucial.

B.12 Turtle Commands

In the framework of scene graphs, turtle commands of L-systems are scene graph nodes. E. g., the turtle command `RU(30)` of the GROGRA software (Sect. 3.15.1 on page 35), which performs a rotation by 30 degrees around the local y-axis, is represented as a transformation node in the scene graph, and the turtle command `F(1)`, which creates a cylinder of length 1 and some given radius, is represented as a node with the geometry of a cylinder and a translation along the axis of the cylinder as transformation. Although the notion of a turtle is uncommon in the context of scene graphs, we still call those scene graph nodes turtle commands which have been adopted from the L-system software GROGRA. They are part of the package `de.grogra.turtle`.

The geometry of some turtle commands of the GROGRA software is only defined when considering the current turtle state. E. g., the sequence of turtle commands `P(15) D(1) F(2)` draws a cylinder of length 2, diameter 1 and colour 15 of the EGA colour palette (i. e., white). So while the length is explicitly specified in a parameter of the symbol, diameter and colour have to be set by preceding turtle commands which modify the turtle state.

To implement this semantics for a scene graph, a derived attribute for the turtle state is defined by the class `de.grogra.turtle.TurtleState`. As for any derived attribute, its value is given by an evaluation along the path (along branch or successor edges) from the root to the node in question. The evaluation looks for nodes being a `de.grogra.turtle.TurtleStateModifier` which are then asked to modify the turtle state according to their semantics. The turtle state is queried by turtle command nodes when the latter need additional information from the state.

Turtle commands for moving or drawing assume that the local z-direction is the growth direction. In fact, this convention is already an intrinsic part of the abstract class `de.grogra.imp3d.objects.AxisBase` defined in the IMP-3D plug-in (Fig. A.5 on page 391). All turtle commands for moving and drawing inherit from this class, but also some geometric primitives like `Cylinder` or `Box`.

Almost all turtle commands of the GROGRA software are defined by the RGG plug-in. Table B.2 gives a short explanation of them, more details can

Turtle Command	Description	GROGRA Notation
`F(x)`	draw cylinder of length `x`	`F(x)`
`F(x, d)`	draw cylinder of length `x` and diameter `d`	`Dl(d) F(x)`
`F(x, d, c)`	draw cylinder of length `x`, diameter `d` and colour `c`	`Pl(c) Dl(d) F(x)`
`F0`	draw cylinder using length of turtle state	`F`
`FAdd(x)`	draw cylinder using length of turtle state, incremented by `x`	`F+(x)`
`FMul(x)`	draw cylinder using length of turtle state, multiplied by `x`	`F*(x)`
`M(x)`, `M0`, `MAdd(x)`, `MMul(x)`	same as above, but only movement	`f(x)`, `f`, `f+(x)`, `f*(x)`
`MRel(q)`	move to relative position `q` on previously created axis	`@(q)`
`RL(a)`, `RU(a)`, `RH(a)`	rotate by `a` degrees around local x/y/z-axis, respectively	`RL(a)`, `RU(a)`, `RH(a)`
`Plus(a)`, `Minus(a)`	rotate by `a`/`-a` degrees around local y-axis	`+`, `-`
`AdjustLU`	rotate around local z-axis such that local y-axis points upwards as far as possible	`$`
`RV(e)`, `RV0`, `RVAdd(e)`, `RVMul(e)`	gravitropism, strength given by `e` and/or turtle state (see `F` above)	`RV(e)`, `RV`, `RV+(e)`, `RV*(e)`
`RG`	maximal gravitropism such that local z-direction points downwards	`RG`
`L(x)`, `L0`, `LAdd(x)`, `LMul(x)`	modify length of turtle state: set to `x`/set to default value/increment by `x`/multiply by `x`	`L(x)`, `L`, `L+(x)`, `L*(x)`
`Ll(x)`, `LlAdd(x)`, `LlMul(x)`	modify local length of turtle state (the length used only for the next `F`)	`Ll(x)`, `Ll+(x)`, `Ll*(x)`
last two rows, but with `C`, `D`, `H`, `N`, `U`, `V` instead of `L`	modify carbon/diameter/heartwood/parameter/internode count/tropism, respectively, of turtle state	
`P(c)`, `P0`, `Pl(c)`	modify colour of turtle state: set to `c`/set to default value/set colour only of next `F` to `c`	`P(c)`, `P`, `Pl(c)`
`OR(o)`	set order of turtle state to `o`	`OR(o)`
`IncScale`	increment scale counter of turtle state	`/`

Table B.2. Turtle commands adopted from GROGRA [103, 106]

Turtle Command	Description
`RD(v, e)`	directional tropism towards direction `v` with strength `e`
`RO(v, e)`	directional tropism towards projection of current moving direction on plane orthogonal to `v` with strength `e`
`RP(p, e)`	positional tropism towards position `p` with strength `e`
`RN(n, e)`	positional tropism towards location of node `n` with strength `e`
`Translate(x, y, z)`	translation by $(\mathtt{x}, \mathtt{y}, \mathtt{z})$
`Rotate(x, y, z)`	rotation by $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ degrees (local coordinate system is rotated around the x-axis first, then around the y-axis and finally around the z-axis)
`Scale(x, y, z)`	scaling by $(\mathtt{x}, \mathtt{y}, \mathtt{z})$
`Scale(s)`	uniform scaling by `s`

Table B.3. Extended turtle commands

be found in [103, 106]. Table B.3 shows extended turtle commands which are not available in the GROGRA software. Furthermore note that all 3D classes of the IMP-3D plug-in shown in Fig. A.5 on page 391 can also be used in the manner of turtle commands as "turtle command" is basically a synonym for "scene graph node" in the new framework.

B.13 Library Functions

The class `de.grogra.rgg.Library` provides a lot of functions and operator overloads as static methods. They are not directly related to the formalism of relational growth grammars, but are useful when specifying typical RGG models. They can roughly be divided into geometric, mathematical, topological and further miscellaneous functions. A lot of the methods are only short helper methods which redirect to other methods spread over the GroIMP API and thus provide easy access to the functionality of GroIMP, tailored for the needs of RGG modelling. As the RGG dialect automatically imports the members of `Library`, they can be addressed by their simple name. In this section, we give a short overview of the library functions, details can be found in the API documentation [101].

B.13.1 Geometric Functions

The geometric functions of `Library` perform several 3D computations which often occur in typical models. All of them are defined in terms of global coordinates which already introduces some kind of global sensitivity (Sect. 3.11

on page 30, Sect. 3.12 on page 32, [103, 106, 155]). `location`, `direction` and `transformation` compute the corresponding 3D properties of a given node using the global coordinate system of the node, where `direction` refers to the local z-direction in global coordinates. Distinguishing the z-direction is by convention: as mentioned in Sect. B.12 on page 406, the local z-direction is the intrinsic growth direction of scene graph nodes extending `de.grogra.imp3d.objects.AxisBase`, which includes the turtle commands for movement and drawing.

The one-parameter method `angle` computes the angle of rotation which a given node implements by its coordinate transformation. The two-parameter method `angle` computes the angle between two given directions. `inclination` returns the angle between the local and global z-axes of a node. `distance` computes the distance between two given nodes.

The two `tropism` methods compute a transformation node which implements a positional or directional tropism, respectively. Contrary to the turtle commands `RP` and `RD`, the methods return a fixed transformation which is not recomputed if the local coordinate system changes.

The next geometric methods use the general framework of the Vecmath plug-in to specify and execute line- and volume-related 3D computations. Within the used framework, volumes are represented by instances of the interface `de.grogra.vecmath.geom.Volume`. The method `distanceToLine` computes the minimal distance of a point to a line, the method `intersect` intersects a line with a volume, and the method `intersectionLength` computes the length of the intersection between a line and a cone. The method `cone` is used to create a cone-shaped volume: for a given node, it uses its location and growth direction for the tip and axis of the cone, and the specified angle determines the half opening angle of the cone. This is a useful method for simple competition models which test for shading or otherwise competing objects in the cone in growth direction. The method `volume` converts any geometric shape into the corresponding representation as a volume. `ray` creates a ray to be used in intersection tests. The two user-defined conversion functions `toPoint3d` and `toTuple3d` return for a given node its location, and the user-defined conversion function `toLine` returns a line which corresponds to the axis of a node, i. e., the connection line between the base point and the tip of a node.

Four constants within 3D space are defined. The vectors `LEFT`, `UP`, `HEAD` are unit vectors of the x-, y-, z-axis, respectively, their names following the convention of turtle geometry. The point `ORIGIN` has the coordinates $(0, 0, 0)$.

B.13.2 Mathematical Functions

Most of the mathematical functions of the library are related to the generation of pseudorandom numbers. `random`, `normal` and `lognormal` compute uniformly, normally and log-normally distributed pseudorandom numbers, respectively. `irandom` is used for discrete uniform pseudorandom numbers,

`distribution` for a discrete distribution with given probabilities. The method `probability` represents a Bernoulli variable with the two outcomes **false**, **true**. Finally, `setSeed` sets the seed for the pseudorandom number generator.

Two aggregate methods `mean` compute the mean value of a sequence of values of `Tuple3f` and `Tuple3d`, respectively. The aggregate method `statistics` computes a statistics of a sequence of values including number of values, mean value and deviation. Of course, also the general filter and aggregate methods of the class `de.grogra.xl.util.Operators` can be used (Sect. 6.3.6 on page 138, Sect. 6.4.3 on page 140), these are automatically imported by the RGG dialect.

The two constants `DEG` and `R2D` can be used as factors to convert from degrees to radians or from radians to degrees, respectively.

B.13.3 Topological Functions

The method `graph` returns the graph of the current model. The constants `successor`, `branch`, `refine`, `contains`, `mark` and `master` are aliases for edge type constants defined elsewhere. `mark` is a general-purpose edge, `master` is used for edges from an `Instance3D` node (Fig. A.5 on page 391) to its master node, i. e., to the node whose geometry shall be instantiated. The constants `EDGE_0` to `EDGE_14` stand for edge types which are not used by GroIMP and may serve as model-specific edge types by declarations like for the neighbourhood edge of the Game of Life in Sect. 10.1.3 on page 277:

```
const int nb = EDGE_0;
```

Most topological functions use axial trees encoded by successor and branch edges as the data structure. The method `ancestor` returns, for a given node, the closest ancestor of a specific type, `successor` the closest successor (on the same axis, i. e., reachable by successor edges). `descendants` yields all descendants of a given type, `minDescendants` a subset thereof, namely only those descendants such that there is no intermediate descendant of the given type. This is similar to the minimal elements variant of transitive closures presented in Sect. 6.5.6 on page 153. Actually, all methods can also be implemented by transitive closures: `ancestor(n, X.class)` is equivalent to `n (: (<-(branch|successor)-)+ X)`, `successor(n, X.class)` is equivalent to `n (: (>)+ X)`, `descendants(n, X.class)` is equivalent to `n (-(branch|successor)->)+ X`, `minDescendants(n, X.class)` is equivalent to `n (-(branch|successor)->)+ : (X)`. The methods can also be used directly as parts of patterns as in `n -ancestor-> X`, in this case the type argument is implicitly set to the required type for the corresponding query variable (Sect. 6.5.3 on page 150). The implementation by transitive closures is more general as we may specify arbitrary path patterns, but the usage of the methods is more efficient with respect to time and stack space. In fact, as the pattern matching algorithm for transitive closures is implemented by recursive invocations (Chap. 7), deep structures may lead to a stack overflow.

Then the library methods have to be used which are implemented without recursive invocations.

The methods `removeLeaves`, `removeTransformationLeaves` as well as `mergeTransformations` and `mergeNonTropismTransformations` serve to simplify the structure without modifying its geometry. The first ones remove leaves from the axial tree which are irrelevant for the further development of the model. E. g., think of rotation nodes which had born a branch until it fell off. Now they are no longer needed and can be removed for the sake of memory savings. The latter methods merge sequences of transformation nodes into a single node with the product of the individual transformations as transformation. This also helps to reduce memory usage.

The method `hide` can be used to hide a subgraph so that is neither displayed nor transformed by rules. Hiding from being displayed is achieved by removing the subgraph from its current parent node and adding it by a mark edge, which is not traversed on display, to the `RGGRoot`. Preventing transformations is achieved by moving all nodes of the subgraph to the last extent list (Sect. B.7 on page 401) which is invisible to queries by default. The method `moveToExtent` sets the extent list to use for a whole subgraph.

The methods `cloneNode` and `cloneSubgraph` clone a single node or a complete subgraph.

B.13.4 Control of Rule Application

Control of rule application can be achieved through the `RGGGraph` instance which implements the graph interface of XL (Sect. B.7 on page 401). This instance is returned by the method `rggGraph`. However, for some methods of this instance there are aliases in the library, namely for `derive` which marks the end of the current parallel production and triggers the corresponding parallel derivation (Sect. 9.1.2 on page 237), for `setDerivationMode` (Sect. 9.1.5 on page 245), and for `allowNoninjectiveMatchesByDefault` which controls the restriction to injective matches (Sect. 9.1.7 on page 250).

The methods `apply` and `applyUntilFinished` return disposable iterators (Sect. 6.13.1 on page 185) to be used in **for**-loops. They allow the repeated execution of code and implicitly invoke `derive` after each iteration to finish the current parallel production. `apply` takes a single argument for the number of iterations, `applyUntilFinished` executes the body as long as possible, i. e., as long as the body gives rise to modifications of the graph.

B.13.5 Creating References to User-Defined Objects

The methods `function`, `curve`, `surface`, `dataset`, `shader`, `image`, `file`, and `reference` return a reference to an object defined within the object explorer of the corresponding type (Sect. A.5 on page 387). This mechanism should be used to refer to such objects from within the code. For example, if the user

has interactively defined a shader named “bark” within the shader explorer, the code should contain a line

```
const Shader barkShader = shader("bark");
```

This reference can be used later on:

```
... Cylinder.(setShader(barkShader)) ...
```

B.13.6 User Interface

Feedback via the graphical user interface is possible in several ways: the static variable `out` is a `PrintWriter` and writes to the XL console panel. But there are also `print`- and `println`- methods directly in the library which also write to the XL console. The content of the status bar can be set with `setStatus`.

GroIMP maintains a current selection of nodes which the user chooses interactively. The currently selected nodes of the graph can be queried by the method `isSelected`. Vice versa, the method `select` can be used to select a set of nodes. This is useful to highlight some relevant nodes which would otherwise be difficult to find for the user.

The methods `plot` and `chart` use the built-in chart feature to plot a function or to show a general dataset. For an example, see Sect. 10.5.5 on page 314.

B.13.7 Operator Methods and Unwrapping Conversions

The library also contains a collection of operator methods. As the RGG dialect automatically imports all methods of the library, these operator overloads are automatically defined.

The operator `<<` is overloaded so that values may be sent to `PrintWriter` instances and to lists. The index operator `[]` is overloaded for list access and function evaluation, but also to access the `i`-th node of a branch. The comparison operators `<`, `>`, `<=`, `>=` define a total order on nodes using their ID.

Furthermore, the class `VecmathOperators`, which is also automatically imported by the RGG dialect, defines useful operator methods for 3D computations: addition, subtraction and multiplication are defined for points, vectors and scalars where meaningful, the operator **`in`** is defined to test whether a point lies within a `Volume`.

The methods `booleanValue`, ..., `doubleValue` are user-defined conversions which unwrap primitive values out of a wrapper node.

B.14 Radiation Model

In general, a radiation model is used to compute the amount of radiation which is absorbed by an object or sensed by a sensor. It takes the geometry of the whole scene, its optical properties and a number of light sources as input.

Exact solutions can only be obtained in very simple situations. Therefore, numerical algorithms have to be used to compute sufficiently precise approximations. For radiation transport, the technique of path tracing turns out to be suitable ([195] and manual of L-Studio software [152]): each light source emits a number of light rays which are then traced through the scene using the optical properties at intersection points to compute a scattered ray. The basic problem of tracing individual light rays, namely diffuse reflection or transmission where the direction after scattering is not a unique function of the incoming direction, is solved by a Monte Carlo method [195], i. e., the new direction is chosen (pseudo-) randomly, or by a Quasi-Monte Carlo method with special regular sequences of numbers instead of pseudorandom numbers [93].

Based on the algorithms of the built-in raytracer Twilight (Sect. A.7.1 on page 392), a radiation model was implemented which takes into account the complete scene of GroIMP. The radiation model is made available by the class `de.grogra.rgg.LightModel`, its usage is easy:

```
const LightModel rad = new LightModel().(setRayCount(1000000));

void step () [
    {radiation.compute();}

    x:Leaf ==>
        ... {float p = rad.getAbsorbedPower(x).integrate(); ...} ... ;
]
```

In the method `step`, at first the radiation model is invoked to compute the amount of radiation absorbed by each geometry node. For this purpose, 1,000,000 primary light rays are shot in total from the light sources. Afterwards, the power absorbed by a node `x` can be obtained by invoking `getAbsorbedPower` on the radiation model with `x` as argument. The result is an instance of the general interface `de.grogra.ray.physics.Spectrum` which represents the spectral distribution of a radiometric quantity. The method `integrate` integrates this distribution to a single value. A variant is the usage of the method `getAbsorbedPower3d` which returns an instance of `de.grogra.ray.physics.Spectrum3d`: this has the variables `x`, `y`, `z` which usually represent the red, green, blue part of the spectrum, respectively. But this can be re-interpreted to stand for arbitrary parts of the spectrum if the specification of the light sources and shaders also uses the RGB channels for other parts of the spectrum than the standard ones.

The radiation model also supports sensors which do not interfere with radiation, but only measure the irradiance at their surface. The sensor class `de.grogra.imp3d.objects.SensorNode` has a disc in the local xy-plane as surface and weights incoming radiation according to the factor $\cos^\gamma \alpha$ where α is the angle of incidence (with respect to the surface normal, i. e., the local z-direction) and γ is an attribute of the node. After computation of the radiation model, the sensed irradiance is queried by the method `getSensedIrradiance`.

B.15 Support for GROGRA Models

The RGG plug-in provides support for GROGRA models: just like `xl`-, `rgg`- and `java`-files of a project are automatically compiled, also `lsy`- and `ssy`-files in the GROGRA syntax are compiled. Sect. 10.5.4 on page 314 shows an example where an existing non-sensitive `lsy`-model was executed and rendered within GroIMP. Currently, not all sensitive functions and methods of GROGRA are supported, but this will be done in the near future.

The compilation of GROGRA source code makes use of the split of the compiler in lexical, syntax and semantic analysis as shown in Fig. 8.2 on page 207. Namely, a specific scanner and parser for the GROGRA syntax were implemented. The result of the parser, the abstract syntax tree, is rearranged and equipped with additional nodes for class and method declarations so that it conforms with an abstract syntax tree of XL source code. Thus, it can immediately be used as input to the XL compiler.

References

1. H. Abelson and A. diSessa. *Turtle Geometry*. MIT Press, Cambridge, Massachusetts, 1982.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
3. V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood, editors. *Document Object Model (DOM) Level 1 Specification*. World Wide Web Consortium, 1998. Available from: `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/` [cited 04 May 2008].
4. Autodesk. Autodesk 3ds Max [online, cited 04 May 2008]. Available from: `http://www.autodesk.com/3dsmax`.
5. Autodesk. Autodesk Maya [online, cited 04 May 2008]. Available from: `http://www.autodesk.com/maya`.
6. G. Barczik and W. Kurth. From designing objects to designing processes: Algorithms as creativity enhancers. In *Predicting the Future. 25th eCAADe Conference Proceedings, Frankfurt/Main, September 26-29, 2007*, pages 887–894, 2007.
7. G. V. Batz, M. Kroll, and R. Geiß. A first experimental evaluation of search plan driven graph pattern matching. In Schürr et al. [174], pages 471–486.
8. M. Bauderon. A uniform approach to graph rewriting: The pullback approach. In M. Nagl, editor, *Graph-Theoretic Concepts in Computer Science*, volume 1017 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 1995.
9. M. Bauderon, H. Jacquet, and R. Klempien-Hinrichs. Pullback rewriting and applications. *Electronic Notes in Theoretical Computer Science*, 51, 2001.
10. U. Bischof. Anbindung der Programmiersprache XL an die 3D-Modelliersoftware Maya über ein Plug-in. Bachelor's thesis, BTU Cottbus, 2006.
11. H. Bisswanger. *Enzymkinetik*. Wiley-VCH, Weinheim, 3rd edition, 2000.
12. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, editors. *XQuery 1.0: An XML Query Language*. World Wide Web Consortium, 2007. Available from: `http://www.w3.org/TR/2007/REC-xquery-20070123/` [cited 04 May 2008].
13. I. A. Borovikov. L-systems with inheritance: An object-oriented extension of L-systems. *SIGPLAN Notices*, 30(5):43–60, 1995.

14. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML progress report. In *Graph Drawing*, pages 501–512, 2001.
15. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, editors. *Extensible Markup Language (XML)*. World Wide Web Consortium, 4th edition, 2006. Available from: `http://www.w3.org/TR/2006/REC-xml-20060816/` [cited 04 May 2008].
16. G. Buck-Sorlin and K. Bachmann. Simulating the morphology of barley spike phenotypes using genotype information. *Agronomie*, 20:691–702, 2000.
17. G. Buck-Sorlin, R. Hemmerling, O. Kniemeyer, B. Burema, and W. Kurth. New rule-based modelling methods for radiation and object avoidance in virtual plant canopies. In *Proceedings of the Second International Symposium on Plant Growth Modeling and Applications, Beijing, November 13-17, 2006*, pages 22–25. IEEE, 2008.
18. G. Buck-Sorlin, R. Hemmerling, O. Kniemeyer, B. Burema, and W. Kurth. A rule-based model of barley morphogenesis, with special respect to shading and gibberellic acid signal transduction. *Annals of Botany*, 101(8):1109–1123, 2008.
19. G. Buck-Sorlin, O. Kniemeyer, and W. Kurth. Barley morphology, genetics and hormonal regulation of internode elongation modelled by a relational growth grammar. *New Phytologist*, 166(3):859–867, 2005.
20. G. Buck-Sorlin, O. Kniemeyer, and W. Kurth. A grammar-based model of barley including virtual breeding, genetic control and a hormonal metabolic network. In Vos et al. [197], pages 243–252.
21. H. Bunke. Programmed graph grammars. In Claus et al. [26], pages 155–166.
22. G. Busatto. *An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation*. PhD thesis, Universität Paderborn, 2002.
23. T. W. Chien and H. Jürgensen. Parameterized L systems for modelling: Potential and limitations. In Rozenberg and Salomaa [169], pages 213–229.
24. J. Clark, editor. *XSL Transformations (XSLT)*. World Wide Web Consortium, 1999. Available from: `http://www.w3.org/TR/1999/REC-xslt-19991116` [cited 04 May 2008].
25. J. Clark and S. DeRose, editors. *XML Path Language (XPath)*. World Wide Web Consortium, 1999. Available from: `http://www.w3.org/TR/1999/REC-xpath-19991116` [cited 04 May 2008].
26. V. Claus, H. Ehrig, and G. Rozenberg, editors. *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*. Springer, 1979.
27. E. S. Coen and E. M. Meyerowitz. The war of the whorls: genetic interactions controlling flower development. *Nature*, 353:31–37, 1991.
28. A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In F. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life, Paris, France*, pages 134–142. Elsevier Publishing, 1992.
29. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Abstract graph derivations in the double pushout approach. In Schneider and Ehrig [171], pages 86–103.
30. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Note on standard representation of graphs and graph derivations. In Schneider and Ehrig [171], pages 104–118.

31. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation – part I: Basic concepts and double pushout approach. In Rozenberg [167], chapter 3, pages 163–246.
32. K. Čulík II and A. Lindenmayer. Parallel rewriting on graphs and multidimensional development. *International Journal of General Systems*, 3:53–66, 1976.
33. K. Čulík II and J. Opatrný. Context in parallel rewriting. In G. Rozenberg and A. Salomaa, editors, *L Systems*, volume 15 of *Lecture Notes in Computer Science*, pages 230–243. Springer, 1974.
34. J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors. *Graph Grammars and their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*. Springer, 1996.
35. R. Dawkins. *The Blind Watchmaker*. Longman, Harlow, 1986.
36. R. Dawkins. The evolution of evolvability. In C. Langton, editor, *Artificial Life, SFI Studies in the Sciences of Complexity*, pages 201–220. Addison-Wesley, Reading, 1988.
37. O. Deussen. *Computergenerierte Pflanzen, Technik und Design digitaler Pflanzenwelten*. Springer, Berlin, 2003.
38. F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In Rozenberg [167], chapter 2, pages 95–162.
39. E. Dufrêne, H. Davi, C. François, G. le Maire, V. le Dantec, and A. Granier. Modelling carbon and water cycles in a beech forest. Part I: Model description and uncertainty analysis on modelled NEE. *Ecological Modelling*, 185:407–436, 2005.
40. Eclipse Foundation. Eclipse.org home [online, cited 04 May 2008]. Available from: `http://www.eclipse.org/`.
41. ECMA. *ECMA-334: C# Language Specification*. European Association for Standardizing Information and Communication Systems, Geneva, Switzerland, 4th edition, 2006.
42. ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. European Association for Standardizing Information and Communication Systems, Geneva, Switzerland, 4th edition, 2006.
43. H. Ehrig. Introduction to the algebraic theory of graph grammars. In Claus et al. [26], pages 1–69.
44. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, Secaucus, NJ, USA, 2006.
45. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, Singapore, 1999.
46. H. Ehrig and A. Habel. Graph grammars with application conditions. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 87–100. Springer, Berlin, 1986.
47. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In Rozenberg [167], chapter 4, pages 247–312.
48. H. Ehrig, M. Korff, and M. Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. Technical Report 90/21, Technische Universität Berlin, 1990.

49. H. Ehrig and H.-J. Kreowski. Parallel graph grammars. In Lindenmayer and Rozenberg [118], pages 425–442.
50. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution*. World Scientific, Singapore, 1999.
51. H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: An algebraic approach. In *IEEE Conference on Automata and Switching Theory*, pages 167–180, Iowa City, 1973.
52. H. Ehrig and G. Rozenberg. Some definitional suggestions for parallel graph grammars. In Lindenmayer and Rozenberg [118], pages 443–468.
53. H. Ehrig and G. Taentzer. From parallel graph grammars to parallel high-level replacement systems. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer Systems*, pages 283–304. Springer, Berlin, 1992.
54. J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In Rozenberg [167], chapter 1, pages 1–94.
55. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In Ehrig et al. [45], pages 551–603.
56. A. P. L. Ferreira and L. Ribeiro. Derivations in object-oriented graph grammars. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *ICGT*, volume 3256 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 2004.
57. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. RFC 2068: Hypertext Transfer Protocol — HTTP/1.1, 1997. Available from: `http://tools.ietf.org/html/rfc2068` [cited 04 May 2008].
58. R. W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8):455–460, 1979.
59. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics. Principles and Practice*. Addison-Wesley, Reading, Massachusetts, 1997.
60. B. Fonkeng. Layout- und Filterverfahren zur Graphdarstellung in GroIMP. Diploma thesis, BTU Cottbus, 2007.
61. W. Fontana. Algorithmic chemistry. In C. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II, SFI Studies in the Sciences of Complexity*, pages 159–209. Addison-Wesley, Reading, 1991.
62. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Boston, San Francisco, New York, 1995.
63. M. Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, 1970.
64. R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2006.
65. R. Geiß and M. Kroll. GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In Schürr et al. [174], pages 568–569.
66. J.-L. Giavitto and O. Michel. MGS: a rule-based programming language for complex objects and collections. *Electronic Notes in Theoretical Computer Science*, 59(4), 2001.
67. C. Godin and Y. Caraglio. A multiscale model of plant topological structures. *Journal of Theoretical Biology*, 191:1–46, 1998.

68. C. Godin, Y. Guédon, E. Costes, and Y. Caraglio. Measuring and analysing plants with the AMAPmod software. In M. Michalewicz, editor, *Plants to Ecosystems: Advances in Computational Life Sciences*, pages 53–84. CSIRO Publishing, Australia, 1997.
69. C. Godin and H. Sinoquet. Functional-structural plant modelling. *New Phytologist*, 166:705–708, 2005.
70. N. S. Goel and I. Rozehnal. Some non-biological applications of L-systems. *International Journal of General Systems*, 18(4):321–405, 1991.
71. N. S. Goel and I. Rozehnal. A high-level language for L-systems and its applications. In Rozenberg and Salomaa [169], pages 231–251.
72. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2005.
73. C. Groer. Dynamisches 3D-Modell der Rapspflanze (*Brassica napus* L.) zur Bestimmung optimaler Ertragskomponenten bei unterschiedlicher Stickstoffdüngung. Diploma thesis, BTU Cottbus, 2006.
74. J. Hanan. *Parametric L-systems and Their Application To the Modelling and Visualization of Plants*. PhD thesis, University of Regina, 1992.
75. T. Heer, D. Retkowitz, and B. Kraft. Algorithm and tool for ontology integration based on graph rewriting. In Schürr et al. [174], pages 577–582.
76. R. Hemmerling, O. Kniemeyer, D. Lanwert, W. Kurth, and G. Buck-Sorlin. The rule-based language XL and the modelling environment GroIMP, illustrated with simulated tree competition. *Functional Plant Biology*, 35:739–750, 2008.
77. M. Henke. Entwurf und Implementation eines Baukastens zur 3D-Pflanzenvisualisierung in GroIMP mittels Instanzierungsregeln. Diploma thesis, BTU Cottbus, 2008.
78. R. Herzog. Ausbau eines bereits implementierten Graphtransformationstools zu einem Plug-in für CINEMA 4D. Bachelor's thesis, BTU Cottbus, 2004.
79. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, 1985.
80. Á. Horváth, G. Varró, and D. Varró. Generic search plans for matching advanced graph patterns. In *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), Braga, Portugal*, Electronic Communications of the EASST, pages 57–68, 2007.
81. T. Huwe. Stochastischer GPU-Strahlenverfolger für GroIMP. Bachelor's thesis, BTU Cottbus, 2008.
82. ISO/IEC. The virtual reality modeling language. ISO/IEC 14772, International Organization for Standardization, Geneva, Switzerland, 1997.
83. ISO/IEC. Programming languages – C++. ISO/IEC 14882, International Organization for Standardization, Geneva, Switzerland, 2003.
84. ISO/IEC. Extensible 3D (X3D). ISO/IEC 19775, International Organization for Standardization, Geneva, Switzerland, 2004.
85. ISO/IEC. Extensible 3D (X3D) encodings. ISO/IEC 19776, International Organization for Standardization, Geneva, Switzerland, 2005.
86. D. Janssens, G. Rozenberg, and R. Verraedt. On sequential and parallel node rewriting graph grammars, part 1. *Computer Graphics and Image Processing*, 18:279–301, 1982.
87. D. Janssens, G. Rozenberg, and R. Verraedt. On sequential and parallel node rewriting graph grammars, part 2. *Computer Graphics and Image Processing*, 23:295–312, 1983.

88. W. Kahl. A relation-algebraic approach to graph structure transformation. In H. C. M. de Swart, editor, *RelMiCS*, volume 2561 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2001.
89. J. T. Kajiya. The rendering equation. In *SIGGRAPH*, pages 143–150, 1986.
90. L. Kari, G. Rozenberg, and A. Salomaa. L systems. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 5, pages 253–328. Springer, Berlin Heidelberg, 1997.
91. R. Karwowski. *Improving the Process of Plant Modeling: The L+C Modeling Language*. PhD thesis, University of Calgary, 2002.
92. R. Karwowski and P. Prusinkiewicz. Design and implementation of the L+C modeling language. *Electronic Notes in Theoretical Computer Science*, 86(2), 2003.
93. A. Keller. *Quasi-Monte Carlo Methods for Photorealistic Image Synthesis*. PhD thesis, University of Kaiserslautern, 1998.
94. J. T. Kim. transsys: A generic formalism for modelling regulatory networks in morphogenesis. In J. Kelemen and P. Sosík, editors, *ECAL 2001*, volume 2159 of *Lecture Notes in Artificial Intelligence*, pages 242–251. Springer, 2001.
95. O. Kniemeyer. The XL language specification [online, cited 04 May 2008]. Available from: `http://www.grogra.de/xlspec/`.
96. O. Kniemeyer. Rule-based modelling with the XL/GroIMP software. In H. Schaub, F. Detje, and U. Brüggemann, editors, *GWAL-6*, pages 56–65, Berlin, 2004. Akademische Verlagsgesellschaft.
97. O. Kniemeyer, G. Barczik, R. Hemmerling, and W. Kurth. Relational growth grammars – a parallel graph transformation approach with applications in biology and architecture. In Schürr et al. [174], pages 152–167.
98. O. Kniemeyer, G. Buck-Sorlin, and W. Kurth. Representation of genotype and phenotype in a coherent framework based on extended L-systems. In W. Banzhaf, T. Christaller, P. Dittrich, J. T. Kim, and J. Ziegler, editors, *ECAL 2003*, volume 2801 of *Lecture Notes in Artificial Intelligence*, pages 625–634. Springer, 2003.
99. O. Kniemeyer, G. Buck-Sorlin, and W. Kurth. A graph-grammar approach to artificial life. *Artificial Life*, 10:413–431, 2004.
100. O. Kniemeyer, G. Buck-Sorlin, and W. Kurth. GroIMP as a platform for functional-structural modelling of plants. In Vos et al. [197], pages 43–52.
101. O. Kniemeyer, R. Hemmerling, and W. Kurth. GroIMP [online, cited 04 May 2008]. Available from: `http://www.grogra.de/`.
102. D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997.
103. 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. Berichte des Forschungszentrums Waldökosysteme, B 38, Göttingen, 1994.
104. W. Kurth. Morphological models of plant growth: Possibilities and ecological relevance. *Ecological Modelling*, 75/76:299–308, 1994.
105. W. Kurth. Some new formalisms for modelling the interactions between plant architecture, competition and carbon allocation. *Bayreuther Forum Ökologie*, 52:53–98, 1998.
106. W. Kurth. *Die Simulation der Baumarchitektur mit Wachstumsgrammatiken*. Wissenschaftlicher Verlag Berlin, 1999.

107. W. Kurth. Spatial structure, sensitivity and communication in rule-based models. In F. Hölker, editor, *Scales, Hierarchies and Emergent Properties in Ecological Models*, volume 6 of *Theorie in der Ökologie*, pages 29–46. Peter Lang, Frankfurt a. M., 2002.
108. W. Kurth and G. Anzola Jürgenson. Triebwachstum und Verzweigung junger Fichten in Abhängigkeit von den beiden Einflußgrößen Beschattung und Wuchsdichte: Datenaufbereitung und -analyse mit GROGRA. In D. Pelz, editor, *Deutscher Verband Forstlicher Forschungsanstalten, Sektion Forstliche Biometrie und Informatik, 10. Tagung Freiburg i. Br. 1997*, pages 89–108. Ljubljana, Biotechnische Fakultät, 1997.
109. W. Kurth, O. Kniemeyer, and G. Buck-Sorlin. Relational growth grammars – a graph rewriting approach to dynamical systems with a dynamical structure. In J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, editors, *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 56–72. Springer, 2004.
110. W. Kurth and B. Sloboda. Growth grammars simulating trees – an extension of L-systems incorporating local variables and sensitivity. *Silva Fennica*, 31:285–295, 1997.
111. S. M. Lane. *Categories for the Working Mathematician*. Springer, New York, 2nd edition, 1998.
112. S. Lang. *Algebra*. Springer, New York, revised 3rd edition, 2002.
113. D. Lanwert. *Funktions-/Strukturorientierte Pflanzenmodellierung in E-Learning-Szenarien*. PhD thesis, University of Göttingen, 2008.
114. L. Lapré. Lparser [online, cited 04 May 2008]. Available from: `http://home.wanadoo.nl/laurens.lapre/lparser.html`.
115. L. Lapré. New Lparser [online, cited 04 May 2008]. Available from: `http://home.wanadoo.nl/laurens.lapre/lparser2.html`.
116. C. Lewerentz. *Interaktives Entwerfen großer Programmsysteme*. PhD thesis, RWTH Aachen, 1988.
117. A. Lindenmayer. Mathematical models for cellular interactions in development. part I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.
118. A. Lindenmayer and G. Rozenberg, editors. *Automata, Languages, Development*. North Holland, Amsterdam, 1976.
119. A. Lindenmayer and G. Rozenberg. Parallel generation of maps: Developmental systems of cell layers. In Claus et al. [26], pages 301–316.
120. T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley, Boston, MA, USA, 2rd edition, 1999.
121. B. Lintermann and O. Deussen. A modelling method and user interface for creating plants. *Computer Graphics Forum*, 17(1):72–82, March 1998.
122. M. Löwe. personal communication, 2007.
123. J. Lück and H. B. Lück. Two-dimensional, differential, intercalary plant tissue growth and parallel graph generating and graph recurrence systems. In Claus et al. [26], pages 284–300.
124. C. M. Macal and M. J. North. Tutorial on agent-based modeling and simulation part 2: how to model with agents. In L. F. Perrone, B. Lawson, J. Liu, and F. P. Wieland, editors, *Winter Simulation Conference*, pages 73–83. WSC, 2006.
125. U. Mannl. Anbindung der Programmiersprache XL an die 3D-Modelliersoftware 3ds max über ein Plug-in. Bachelor's thesis, BTU Cottbus, 2006.

126. MAXON. CINEMA 4D [online, cited 04 May 2008]. Available from: http://www.maxon.net/.
127. B. Mayoh. Another model for the development of multidimensional organisms. In Lindenmayer and Rozenberg [118], pages 469–486.
128. Microsoft Corporation. C# version 3.0 specification. Technical report, Microsoft Corporation, 2005. Available from: http://msdn.microsoft.com/netframework/future/linq/ [cited 04 May 2008].
129. R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH*, pages 397–410, 1996.
130. M. Nagl. On a generalization of Lindenmayer-systems to labelled graphs. In Lindenmayer and Rozenberg [118], pages 487–508.
131. M. Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierungen*. Vieweg, Braunschweig, 1979.
132. M. Nagl. A tutorial and bibliographical survey on graph grammars. In Claus et al. [26], pages 70–126.
133. M. Nagl, A. Schürr, and M. Münch, editors. *AGTIVE'99 International Workshop on Applications of Graph Transformation with Industrial Relevance*, volume 1779 of *Lecture Notes in Computer Science*. Springer, Berlin, 2000.
134. N. T. Nikolov, W. J. Massman, and A. W. Schoettle. Coupling biochemical and biophysical processes at the leaf level: an equilibrium photosynthesis model for leaves of C_3 plants. *Ecological Modelling*, 80:205–235, 1995.
135. K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and C. Jacobi. Pascal-P implementation notes. In D. W. Barron, editor, *Pascal – The Language and its Implementation*, pages 125–170. John Wiley, 1981.
136. Object Management Group. Unified modeling language [online, cited 04 May 2008]. Available from: http://www.omg.org/technology/documents/formal/uml.htm.
137. Object Refinery Limited. JFreeChart [online, cited 04 May 2008]. Available from: http://www.jfree.org/jfreechart/.
138. ObjectWeb Consortium. ASM – Home Page [online, cited 04 May 2008]. Available from: http://asm.objectweb.org/.
139. F. Parisi-Presicce, H. Ehrig, and U. Montanari. Graph rewriting with unification and composition. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Third International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 496–514. Springer, 1987.
140. T. J. Parr. ANTLR parser generator [online, cited 04 May 2008]. Available from: http://www.antlr.org.
141. T. J. Parr. *Obtaining practical variants of LL (K) and LR (K) for K greater than 1 by splitting the atomic K-tuple*. PhD thesis, Purdue University, West Lafayette, IN, USA, 1993.
142. M. Pastorová. A comparison of two controlled rewriting mechanisms for table Lindenmayer systems. In Rozenberg and Salomaa [169], pages 183–189.
143. A. Paz. Multidimensional parallel rewriting systems. In Lindenmayer and Rozenberg [118], pages 509–515.
144. H.-O. Peitgen, H. Jürgens, and D. Saupe. *Bausteine des Chaos – Fraktale*. Springer-/Klett-Cotta Verlag, Heidelberg, Stuttgart, 1992.
145. Perl Foundation. The Perl directory – perl.org [online, cited 04 May 2008]. Available from: http://www.perl.org/.

146. J. L. Pfaltz. Web grammars and picture description. *Computer Graphics and Image Processing*, 1:193–220, 1972.
147. J. L. Pfaltz, M. Nagl, and B. Böhlen, editors. *Applications of Graph Transformations with Industrial Relevance, AGTIVE 2003, Revised Selected and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*. Springer, 2004.
148. J. Pfreundt and B. Sloboda. The relation of local stand structure to photosynthetic capacity in a spruce stand: a model calculation. *Lesnictví-Forestry*, 42:149–160, 1996.
149. L. Piegl and W. Tiller. *The NURBS book*. Springer, 2nd edition, 1997.
150. J. L. Prioul and P. Chartier. Partitioning of transfer and carboxylation components of intracellular resistance to photosynthetic CO_2 fixation: A critical analysis of the methods used. *Annals of Botany*, 41:789–800, 1977.
151. PROGRES. Research: PROGRES [online, cited 04 May 2008]. Available from: `http://www-i3.informatik.rwth-aachen.de/research/progres/`.
152. P. Prusinkiewicz. Algorithmic botany: Home [online, cited 04 May 2008]. Available from: `http://www.algorithmicbotany.org/`.
153. P. Prusinkiewicz. Graphical applications of L-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 247–253, Toronto, Ont., Canada, 1986. Canadian Information Processing Society.
154. P. Prusinkiewicz. Modeling plant growth and development. *Current Opinion in Plant Biology*, 7(1):79−83, 2004.
155. P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Měch. Visual models of plant development. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 535–597. Springer, Berlin, 1997.
156. P. Prusinkiewicz, M. Hammel, and E. Mjolsness. Animation of plant development. In *SIGGRAPH*, pages 351–360, 1993.
157. P. Prusinkiewicz and J. Hanan. L-systems: from formalism to programming languages. In Rozenberg and Salomaa [169], pages 193–211.
158. P. Prusinkiewicz and L. Kari. Subapical bracketed L-systems. In Cuny et al. [34], pages 550–564.
159. P. Prusinkiewicz, R. Karwowski, and B. Lane. The L+C plant modelling language. In Vos et al. [197], pages 27–42.
160. P. Prusinkiewicz, R. Karwowski, J. Perttunen, and R. Sievänen. Specification of L – a plant-modeling language based on Lindenmayer systems. Version 0.5. Research note, University of Calgary, Department of Computer Science, 1999.
161. P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, New York, 1990.
162. Python Software Foundation. Python programming language – official website [online, cited 04 May 2008]. Available from: `http://www.python.org/`.
163. A. Rensink, A. Dotor, C. Ermel, S. Jurack, O. Kniemeyer, J. de Lara, S. Maier, T. Staijen, and A. Zündorf. Ludo: A case study for graph transformation tools. In Schürr et al. [174], pages 493–513.
164. A. Rensink and G. Taentzer. AGTIVE 2007 graph transformation tool contest. In Schürr et al. [174], pages 487–492.
165. S. Rogge. Generierung von Baumdarstellungen in VRML für den Branitzer Park. Bachelor's thesis, BTU Cottbus, 2008.
166. G. Rozenberg. T0L systems and languages. *Information and Control*, 23(4):357–381, 1973.
167. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

168. G. Rozenberg and A. Salomaa, editors. *The Book of L*. Springer, Berlin, 1986.
169. G. Rozenberg and A. Salomaa, editors. *Lindenmayer Systems*. Springer, Berlin, 1992.
170. RULE. Rule 2007 [online, cited 04 May 2008]. Available from: `http://www.lsv.ens-cachan.fr/rdp07/rule.html`.
171. H. J. Schneider and H. Ehrig, editors. *Graph Transformations in Computer Science, International Workshop, Dagstuhl Castle, Germany, January 1993, Proceedings*, volume 776 of *Lecture Notes in Computer Science*. Springer, 1994.
172. S. Schneider. Konzeption eines Subsystems für die GroIMP-Plattform sowie eines zugrundeliegenden XML-Datenformats zum Austausch graphbasierter, multiskalierter Strukturen. Diploma thesis, BTU Cottbus, 2006.
173. A. Schürr. Programmed graph replacement systems. In Rozenberg [167], chapter 7, pages 479–546.
174. A. Schürr, M. Nagl, and A. Zündorf, editors. *Applications of Graph Transformations with Industrial Relevance, International Workshop, AGTIVE '07*, volume 5088 of *Lecture Notes in Computer Science*. Springer, 2008.
175. A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [45], pages 487–550.
176. A. Skusa, W. Banzhaf, J. Busch, P. Dittrich, and J. Ziegler. Künstliche Chemie. *Künstliche Intelligenz*, 1/00:12–19, 2000.
177. C. Smith. *On Vertex-Vertex Systems and Their Use in Geometric and Biological Modelling*. PhD thesis, University of Calgary, 2006.
178. C. Smith, P. Prusinkiewicz, and F. F. Samavati. Local specification of surface subdivision algorithms. In Pfaltz et al. [147], pages 313–327.
179. K. Smoleňová and R. Hemmerling. Growing virtual plants for virtual worlds. In *Proceedings of the 24th Spring Conference on Computer Graphics, April 21-23, 2008, Budmerice Castle, Slovakia*, 2008.
180. M. Steilmann. *Morphologische Untersuchungen zur Modellierung des Wachstums in Abhängigkeit von den Licht- und Konkurrenzverhältnissen von Jungbuchen*. Diploma thesis, University of Göttingen, 1996.
181. G. Stiny and J. Gips. Shape grammars and the generative specification of painting and sculpture. In *IFIP Congress 1971*, pages 125–135. North Holland Publishing, 1971.
182. J. Strobel. *Die Atmumg der verholzten Organe von Altbuchen (*Fagus sylvativa *L.) in einem Kalk- und einem Sauerhumusbuchenwald*. PhD thesis, University of Göttingen, 2004.
183. Sun Microsystems. JAR file specification [online, cited 04 May 2008]. Available from: `http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html`.
184. Sun Microsystems. Java Data Objects (JDO) [online, cited 04 May 2008]. Available from: `http://java.sun.com/jdo/`.
185. Sun Microsystems. Java Native Interface Specification [online, cited 04 May 2008]. Available from: `http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html`.
186. Sun Microsystems. KSL: Kitchen sink language [online, cited 04 May 2008]. Available from: `http://openjdk.java.net/groups/compiler/ksl.html`.
187. Sun Microsystems. Advantages of the OpenOffice.org XML file format used by the StarOffice office suite. White paper, Sun Microsystems, 2004. Available from: `http://www.sun.com/software/star/staroffice/7/whitepapers/index.xml` [cited 04 May 2008].

188. J. Szuba. New object-oriented PROGRES for specifying the conceptual design tool GraCAD. In T. Mens, A. Schürr, and G. Taentzer, editors, *Electronic Notes in Theoretical Computer Science*, volume 127, pages 141–156, 2005.
189. J. Szuba, A. Ozimek, and A. Schürr. On graphs in conceptual engineering design. In Pfaltz et al. [147], pages 75–89.
190. J. Szuba, A. Schürr, and A. Borkowski. GraCAD – graph-based tool for conceptual design. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *ICGT*, volume 2505 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2002.
191. G. Taentzer, E. Biermann, D. Bisztray, B. Bohnet, I. Boneva, A. Boronat, L. Geiger, R. Geiß, Á. Horvath, O. Kniemeyer, T. Mens, B. Ness, D. Plump, and T. Vajk. Generation of Sierpinski triangles: A case study for graph transformation tools. In Schürr et al. [174], pages 514–539.
192. TU Berlin. The AGG homepage [online, cited 04 May 2008]. Available from: `http://tfs.cs.tu-berlin.de/agg/`.
193. D. Varró, M. Asztalos, D. Bisztray, A. Boronat, D.-H. Dang, R. Geiß, J. Greenyer, P. V. Gorp, O. Kniemeyer, A. Narayanan, E. Rencis, and E. Weinell. Graph transformation tools contest on the transformation of UML models to CSP. In Schürr et al. [174], pages 540–565.
194. G. Varró, Á. Horváth, and D. Varró. Recursive graph pattern matching: With magic sets and global search plans. In Schürr et al. [174], pages 456–470.
195. E. Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1998.
196. H. von Koch. Une méthode géométrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes. *Acta Mathematica*, 30:145–174, 1906.
197. J. Vos, L. F. M. Marcelis, P. H. B. de Visser, P. C. Struik, and J. B. Evers, editors. *Functional-Structural Plant Modelling in Crop Production, International Workshop*, volume 22 of *Wageningen UR Frontis Series*. Springer, 2007.
198. S. Wolfram. *A new kind of science*. Wolfram Media, Champaign, Illinois, 2002.
199. D. Zhao. Simulation und Visualisierung der Struktur und Dynamik metabolischer Netzwerke mit relationalen Wachstumsgrammatiken. Diploma thesis, BTU Cottbus, 2006.
200. A. Zündorf. Graph pattern matching in PROGRES. In Cuny et al. [34], pages 454–468.
201. A. Zündorf and A. Schürr. Nondeterministic control structures for graph rewriting systems. In G. Schmidt and R. Berghammer, editors, *Graph-Theoretic Concepts in Computer Science, 17th International Workshop, WG'91, Fishbachau, Germany*, number 643 in Lecture Notes in Computer Science, pages 48–62. Springer, 1991.

Index