Extension of the Rule-Based Programming Language XL by Concepts for Multi-Scaled Modelling and Level-of-Detail Visualization

Dissertation
for the award of the degree
"Doctor rerum naturalium" (Dr.rer.nat.)
of the Georg-August-Universität Göttingen

within the doctoral program Environmental Informatics (PEI)
of the Georg-August University School of Science (GAUSS)

submitted by
Yongzhi Ong

from Singapore
Göttingen, 2015

Thesis Committee

Prof. Dr. Winfried Kurth
(Department Ecoinformatics, Biometrics & Forest Growth, Georg-August University of Göttingen)
Prof. Dr. Joachim Saborowski
(Department Ecoinformatics, Biometrics & Forest Growth, Georg-August University of Göttingen)

Members of the Examination Board
Reviewer: **Prof. Dr. Winfried Kurth**
(Department Ecoinformatics, Biometrics & Forest Growth, Georg-August University of Göttingen)
Second Reviewer: **Prof. Dr. Hans-Jörg Kreowski**
(Department of Computer Science, University of Bremen)

Further members of the Examination Board

Prof. Dr. Holger Kreft
(Biodiversity, Macroecology & Conservation Biogeography Group, Georg-August University of Göttingen)
Prof. Dr. Joachim Saborowski
(Department Ecoinformatics, Biometrics & Forest Growth, Georg-August University of Göttingen)
Prof. Dr. Anita Schöbel
(Research Group Optimization, Institute for Numerical and Applied Mathematics, Georg-August University of Göttingen)
Prof. Dr. Stephan Waack
(Theoretical Computer Science and Algorithmic Methods, Georg-August University of Göttingen)

Date of the oral examination: 27.04.2015

Gedruckt mit Unterstützung des Deutschen Akademischen Austauschdienstes.
Printed with the support of the German Academic Exchange Service.

Acknowledgements

I would like to thank:

- Prof. Dr. Winfried Kurth - for his liberal guidance, immense patience, and bringing me into this special domain of science.
- Prof. Dr. Joachim Saborowski - for supporting my research work in Göttingen.
- Prof. Dr. Hans-Jörg Kreowski - for taking an interest in this dissertation.
- Mum and Dad - for love and support.
- Grandma - who walked me to the door and opened another.
- Chew Geik - for the sweet and bitter memories we share.
- Yi and Si Jie - for taking care of the family while I am away.
- Katarina - for her friendship, advice, and lending a listening ear during the darker days.
- Gunnar, Michael, Reinhard - for being friends and partners in research.
- Hien and Dit Dit - for being friends and adding joy to time spent in our department.
- Liz and Kevin - for giving a tinge of home in a faraway place.
- Ole - for the creation that made this dissertation possible.
- Ms. Ilona Watteler-Sprang and Dr. Reinhold Meyer - for the wonderful administrative and technical support in our department.
- Everyone whom I met in this journey towards a PhD degree.

Contents

Page

1 Introduction & Motivation **1**
1.1 Programming Languages 1
1.1.1 From Mechanical Calculators to Electronic Computers 1
1.1.2 From Logic to Algorithms 3
1.1.3 Generations of Programming Languages 4
1.2 The XL Programming Language 5
1.3 Computer Graphical Modelling & Level-of-Detail (LOD) Visualization 7
1.4 Motivations 8
1.4.1 Research Questions 9
1.4.2 Research Objectives 10

I The Rule-Based Paradigm & XL **12**

2 Linear Rewriting Systems **13**
2.1 Introduction to Formal Languages 13
2.1.1 Alphabets, Words & Languages 14
2.1.2 The Chomsky Grammars and Hierarchy 15
2.1.2.1 Phrase-structure Grammar (type 0) 15
2.1.2.2 Context-sensitive Grammar (type 1) 16
2.1.2.3 Context-free Grammar (type 2) 16
2.1.2.4 Regular Grammar (type 3) 16
2.2 L-systems 17
2.2.1 0L-systems & D0L-systems 18
2.2.2 E0L-systems 20
2.2.3 T0L-systems & DT0L-systems 20
2.2.4 Bracketed L-systems 21

2.2.5 Context-sensitive L-systems . . . 24
2.2.6 Stochastic L-systems . . . 25
2.2.7 Pseudo L-systems . . . 26
2.2.8 Extending L-systems from Discrete to Continuous . . . 27
2.2.8.1 Parametric L-systems . . . 27
2.2.8.2 Differential L-systems . . . 28
2.2.9 L-systems with Global and External Interactivity . . . 29
2.2.9.1 Growth Grammars . . . 29
2.2.9.2 Environmentally-sensitive L-systems . . . 30
2.2.9.3 Open L-systems . . . 30

3 Graph Rewriting **31**
3.1 Introduction to Graphs . . . 32
3.1.1 Fundamental Definitions . . . 32
3.1.1.1 Alphabet . . . 32
3.1.1.2 Graph . . . 32
3.1.1.3 Subgraph . . . 32
3.1.1.4 Graph Homomorphism . . . 33
3.1.1.5 Partial Graph Homomorphism . . . 34
3.1.2 Graphs in Category Theory . . . 35
3.1.2.1 Category . . . 35
3.1.2.2 Monomorphism . . . 36
3.1.2.3 Epimorphism . . . 37
3.1.2.4 Isomorphism . . . 38
3.1.2.5 Functor . . . 38
3.1.2.6 Category-of-Paths . . . 39
3.1.2.7 Diagram-in-Category . . . 41
3.1.2.8 Cone & Cocone . . . 42
3.1.2.9 Limit & Colimit . . . 45
3.1.2.10 Product & Coproduct . . . 48
3.1.2.11 Equalizer & Coequalizer . . . 50
3.1.2.12 Pullback & Pushout . . . 53
3.2 Fundamentals of Graph Rewriting . . . 55
3.2.1 The Double-Pushout Approach (DPO) . . . 56
3.2.2 The Single-Pushout Approach (SPO) . . . 58
3.2.3 Neighbourhood Controlled Embedding . . . 61
3.3 Graph Rewriting in XL . . . 66
3.3.1 Parallel Single-Pushout (SPO) Approach . . . 66

3.3.2 L-system-style Connection 68
3.3.2.1 Operator-based Graph Rewriting 69
3.3.2.2 Operator-based L-system-style Graph Rewriting 70
3.3.2.3 Single-pushout (SPO) with Operators 75

4 XL for Multiscale Modelling 81
4.1 Multiscale Modelling Framework 82
4.2 Statement of Problem 85
4.3 Multiscale Graph Data Structure 89
4.3.1 Structure-of-Scales 89
4.3.2 Type Graph 91
4.3.3 Instanced Graph 93
4.4 Multiscale Graph Rewriting 93
4.4.1 Scale-specific L-system-style Connection 95
4.4.2 Multiscale Connection 100
4.4.2.1 Partial Multiscale Embedding 102
4.4.2.2 Total Multiscale Embedding 106
4.4.2.3 On Parallelism in Multiscale Embedding . . . 111
4.5 XL Multiscale Syntax & Features 115
4.5.1 Syntax Extensions 115
4.5.2 The Observer Programming Pattern 120
4.6 Technical Documentation 127
4.6.1 Use Cases 127
4.6.2 Compilation 129
4.6.3 Run 130
4.6.4 Visualization and the Observer Pattern 132

II Level-of-Detail (LOD) Visualization 137

5 Multiscale & LOD Visualization 138
5.1 Incremental LOD for Branching Structures 138
5.1.1 Previous Work 139
5.1.2 Polyline Incremental LOD and Ramification LOD . . . 141
5.1.2.1 Polyline Incremental LOD 142
5.1.3 Ramification LOD 144
5.2 The Multiscale Graph and Grammar 144

5.2.1 Local - The Multiscale Branching Structure 145
5.2.2 Global - The Multiscale Scene Graph 148
5.2.3 Update and Extraction for Rendering 149
5.3 Implementation and Results 150
5.4 Summary and Future Work 150

III Applications & Examples 159

6 Examples and Demonstrative Models 160
6.1 Fission Yeast 160
6.1.1 Single Cell Model 160
6.1.2 Multiple Cell Model 163
6.1.3 Rule-based Species and Complexes 165
6.2 Beech Structural Growth 166
6.3 Specifying and Generating a Multiscale Plant Structure 169
6.4 Crown Generation 173
6.5 *Fagus sylvatica* Stand under Ozone Exposure 178
6.6 Stand Dynamics and Morphological Developments of Conifers 189
6.6.1 Multiscale Graph Structure and Model Initialization . . 189
6.6.2 Germination 191
6.6.3 Growth 192
6.6.3.1 Stand Competition Index 192
6.6.3.2 Individual Tree Growth 193
6.6.3.3 Structural and Architectural Development . . 195
6.6.4 Mortality 198

7 Conclusion 200
7.1 Answers to Research Questions 200
7.2 Concluding Remarks 203

8 Appendix: Interfaces with Other Software 204
8.1 The MTG File Interface 204
8.2 The Xplo and ArchiTree Interface 207

List of Tables

6.1 Multiscale framework for crown generation 175
6.2 Simulation steps for beech stands under ozone exposure 184
6.3 Parameter values for multiscale stand dynamics 199

List of Figures

2.1 Tree diagram of T0L-system derivations 21
2.2 Turtle interpretation of bracketed D0L-system derivations . . . 24
2.3 Turtle interpretation of context-sensitive 1L-system modelling acropetal signal propagation 26
2.4 Sierpiński "arrowhead" based on turtle interpretation of pseudo L-system 27
2.5 Turtle interpretation of parametric 0L-system 28

3.1 Graph homomorphism 34
3.2 Not a graph homomorphism 34
3.3 Monomorphism in the category Graph 37
3.4 Epimorphism in the category Graph 38
3.5 Isomorphism in the category Graph 39
3.6 Object and arrow functions of a functor 40
3.7 A category and its corresponding category-of-paths 41
3.8 Diagram-in-Category: $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$ 42
3.9 Categories forming bases of cones and cocones 44
3.10 Cone 45
3.11 Cocone 45
3.12 Limit 47
3.13 Colimit 48
3.14 Categorical product with apex K_2 49
3.15 Categorical product with detailed graph homomorphisms . . . 50
3.16 Categorical coproduct with apex K_2 51
3.17 Categorical coproduct with detailed graph homomorphisms . . 52
3.18 Coequalizer 53
3.19 Categorical pushout with detailed graph homomorphisms . . . 55
3.20 Graph rewrite 57
3.21 DPO graph rewrite 59

3.22 DPO identification condition failure 60
3.23 DPO dangling condition failure 61
3.24 SPO production and match 62
3.25 SPO step 1 - gluing 63
3.26 SPO step 2 - deletion 64
3.27 SPO direct derivation overview 64
3.28 edNCE graphs and production 65
3.29 Derivation of edNCE production 66
3.30 Parallel SPO derivation: step 2 69
3.31 Derivation of production with operators 71
3.32 Graph translation of well-nested word 72
3.33 L-system-style production 73
3.34 SPO derivation with operators 79

4.1 Multiscale tree graph (MTG) [67] 82
4.2 Multiscale scale integration steps 85
4.3 L-system style embedding with multiple scales 87
4.4 SPO embedding with multiple scales 88
4.5 Refinement orderings 89
4.6 Structure-of-Scales and Type Graph 90
4.7 Instanced Graph 94
4.8 Scale specific L-system-style connection setup 97
4.9 Scale specific L-system-style production 97
4.10 Scale specific L-system-style derived graph 99
4.11 Operators for multi-scale embedding 102
4.12 Partial multi-scale embedding setup 103
4.13 Partial multi-scale embedding production 104
4.14 Partial multi-scale embedding derived graph 105
4.15 Total multi-scale embedding setup 108
4.16 Total multi-scale embedding production 109
4.17 Total multi-scale embedding derived graph 111
4.18 Prevention of embedding edges between adjacent matches . . . 113
4.19 XL multi-scale syntax without branching 118
4.20 Query or production graph with cycles 120
4.21 Syntax for rule establishing refinement embedding edges - 1 . . 121
4.22 Syntax for rule establishing refinement embedding edges - 2 . . 121
4.23 Syntax for rule establishing refinement embedding edges - 3 . . 122
4.24 Half-edge matching 123

4.25 Use case diagram for user of GroIMP 128
4.26 Class diagram for classes used in compilation 134
4.27 Class diagram for classes modified for running multiscale XL code 135
4.28 Class diagram for multiscale visualization 136

5.1 Polyline and simplification line with enclosed area 142
5.2 Polyline and simplification line with incremental term 143
5.3 Ramification of branching line 145
5.4 Graph and geometric interpretation 146
5.5 Illustration of type graph 147
5.6 Applying the multiscale rule for 2 time steps 152
5.7 Multiscale graph representation of complete branching structure 153
5.8 Illustrations of scene graphs 154
5.9 Pseudo-code for the first type of graph traversal 155
5.10 Pseudo-code for the second type of graph traversal 155
5.11 Pseudo-code for selective traversal of branching structure . . . 156
5.12 Results of simulation tree growth for 40 time steps 157
5.13 Multiple resolutions of branching structure 158

6.1 Fission yeast cell division simulation structure of scales 161
6.2 Fission yeast cell division simulation model graph 162
6.3 Screenshot of cell division process visualization 165
6.4 Species and complexes 166
6.5 Beech model structure of scales and type graph 167
6.6 Beech structural growth 168
6.7 Beech structural growth 168
6.8 Illustration of graph structures for multiscale plant structure . 169
6.9 Initial instanced graph modified by an XL rule 171
6.10 Illustration of graph structures for modelling crown layers . . . 174
6.11 Illustration of multiscale crown model development 176
6.12 Modification of instanced graph at organ scale 177
6.13 Multiscale vs. single scale crown generation 179
6.14 Bud count comparison between multiscale and single-scale models 180
6.15 Simulation time comparison between multiscale and single-scale models 180

6.16 Illustration of graph structures for beech stands under ozone exposure . . . 182
6.17 Illustration of initial instanced graph for beech stands . . . 183
6.18 Application of multiscale framework for beech stands under ozone exposure . . . 185
6.19 Example of an execution of a multiscale rule in XL . . . 188
6.20 Screenshot of the three beech stands after sixteen simulation years . . . 189
6.21 Graph construct for model simulating stand dynamics . . . 190
6.22 Illustration of tree variables . . . 197
6.23 Illustration of multiscale tree stand . . . 199

8.1 MTG interface class diagram . . . 206
8.2 Visualization of imported MTG in GroIMP . . . 208
8.3 ArchiTree and multiscale instanced graph . . . 209
8.4 Class diagram of GroIMP module in Xplo . . . 211
8.5 Screenshot of Xplo displaying GroIMP generated trees . . . 212

Listings

4.1 XL Module Declaration 115
4.2 XL Fundamental Methods 116
4.3 XL Host and Type Graph Construction 116
4.4 XL Multiscale Rule 119
4.5 Structure of scales construction 124
4.6 Establishment of observer pattern in XL 125
6.1 XL Type Graph Construction 169
6.2 XL Type Graph Construction for beech stand 183
6.3 Specifying enzyme concentrations using ozone conditions in tree stand 186
6.4 Reducing photosynthetic production of leaves based on crown layer capacity 186
6.5 Beech growth with multiple scales 187
6.6 Creation of grid floor for forest stand 190
6.7 Creation of type graph for forest stand 190
6.8 Germination rule 191
6.9 Grid to tree refinement 191
6.10 Sum of dbh for competition index computation 192
6.11 Setting competition index for trees 193
6.12 Multiscale tree development based on competition indices 193
6.13 Tree trunk development 195
6.14 First order branch development 195
6.15 Mechanical bending 196
6.16 Mortality 198

Chapter 1

Introduction & Motivation

The effect of scale depends not on a thing itself, but in relation to its whole environment or milieu; it is in conformity with the thing's 'place in Nature', its field of action and reaction in the Universe. - D'Arcy Thompson, 1952.

1.1 Programming Languages

The idea of a language involves people, communication, and any system of symbols or signs [114]. When speaking of programming languages, however, communication generally refers to that from man to machine.

1.1.1 From Mechanical Calculators to Electronic Computers

One of the earliest machines for computation, or more specifically, calculation, was invented by Blaise Pascal in 1642 [24]. The machine, known as a Pascaline, was a mechanical device that could add and subtract two numbers. Input to Pascal's calculator was given by rotating wheels, each representing a digit zero through nine. Around 1672, Gottfried Wilhelm Leibniz invented the Stepped Reckoner, the first calculator that could perform all four arithmetic operations (interestingly, one of the machines was discovered in an attic at the end of the 19th century in the University of Göttingen) [110]. Input to the Stepped Reckoner was given by rotating dials, in a manner similar to the Pascaline, and a crank to perform the calculations. Historically, tables such as those for logarithmic and trigonometric functions were tabulated by peo-

ple, often resulting in errors. In 1821, Charles Babbage attempted to resolve the problem by designing and proposing a mechanical calculator, the Difference Engine, which could tabulate polynomial functions [132]. While doing so, he realized that a design for a general purpose computer was possible and consequently proposed the Analytical Engine in 1834. The Analytical Engine had an arithmetic logic unit, control flow, and integrated memory. Hence, input of programs in addition to data was necessary. Babbage utilized the idea of punch cards, first demonstrated by Joseph Marie Jacquard in 1801 to direct mechanical looms [78], for providing input to the Analytical Engine. Three different types of punch cards were used: one for arithmetical operations [18, 19], one for numerical constants, and one for load and store operations, transferring numbers from the store to the arithmetic unit or back. The programming language employed in the design was similar to modern assembly language. A description of the Engine was written by Luigi Menabrea in 1842 [118]. In 1843, while translating Menabrea's description to English, Ada Lovelace appended a way to calculate Bernoulli numbers using the engine and punch cards, becoming the first computer programmer. On a similar note, in 1890, Herman Hollerith invented the tabulating machine that read punch cards for processing the 1890 United States census [80].

Meanwhile, strides were taken towards the expression of logic with mathematics. Leibniz wrote about his Calculus Ratiocinator, a theoretical universal logical calculation framework. Some understand it as the beginning of symbolic logic [99]. In 1847, George Boole published a pamphlet titled "Mathematical Analysis of Logic" [12]. The article developed ideas on a logical method, and he argued that logic should be a branch of mathematics instead of a part of philosophy. Boole showed how Aristotle's syllogistic logic could be expressed as algebraic equations and published further refinements to his ideas later in his book, "An Investigation of the Laws of Thought" [13]. These concepts have come to be known collectively as Boolean algebra. Some perceive Boole's work as the origin of modern logic [54]. Boole's work was not of particular interest to many until Claude Shannon employed it to simplify the design of circuits and telephone routing switches [159]. Shannon's master's thesis became the foundation for digital circuit design, laying grounds for the development of modern electronic computers.

1.1.2 From Logic to Algorithms

Another significant event in the history of programming languages occurred in 1879, when Gottlob Frege published his "Begriffsschrift" [57]. In this publication, Frege showed a formal system together with propositions, axioms, universal and existential quantification, and formalism of proof [132]. This formal system became the basis of modern predicate logic. Frege's motivation for this work resembled Leibniz's motivation towards a universal logical calculation framework and universal conceptual language. He intended to show that mathematics was reducible to logic and attempted to derive all the laws of arithmetic from axioms he asserted as logical [58]. However, in 1903, Bertrand Russell proved one of the axioms inconsistent (the inconsistency is known as Russell's paradox)[152]. Several alternatives were proposed to resolve the inconsistency but the details are omitted here.

Russell's paradox was met by David Hilbert's programme in the early twentieth century. His programme was concerned with the formalization of mathematics, i.e. the axiomatization of mathematics, and proof that the axiomatization was consistent. In 1900, he posed a set of twenty-three key problems that needed to be solved [121]. The problems were not just mathematical problems but problems about the formalization of mathematics itself. Taken together, the problems can be summarized in three questions:

- Is mathematics complete? That is, can every mathematical statement be proved or disproved from a given set of axioms?
- Is mathematics consistent? That is, can only true statements be proved?
- Is every statement in mathematics decidable? That is, is there an algorithmically computable function that can be applied to every statement to determine if the statement is true or false in finite time?

The first two questions were addressed by Kurt Gödel the same year (and at the same conference) they were presented by Hilbert. Gödel proved that if mathematics is consistent, then mathematics is not complete. That last question is known by its German name as the *Entscheidungsproblem*. It was considered by Leibniz when he was building a mechanical calculator, and wondered if a machine that could determine if particular mathematical statements are true or false was possible. In 1935, Alan Turing was introduced to Gödel's incompleteness theorem while studying under Max Newman. When he understood Gödel's results, Turing was able to see the

answer to the Entscheidungsproblem as "no" [121]. He began to define the notion of an algorithm with a similar intuition as Leibniz more than two centuries earlier. Turing formulated his definition by thinking about a powerful calculating machine that could not only perform arithmetic but also could manipulate symbols in order to prove mathematical statements. The mental design of such a machine is now known as a Turing machine. In 1936, Turing, and independently, Alonzo Church, published papers [170, 29] showing that a solution to the Entscheidungsproblem is impossible. Their assertions were based on the assumption that a function is algorithmically computable if and only if it is computable by a Turing machine, or equivalently, expressible by lambda calculus. Although a formal definition of an algorithm remains a challenging problem, Turing machines were put forth as the definition [122]. Today, programmers are essentially writing algorithms using programming languages.

1.1.3 Generations of Programming Languages

Before describing the evolution of programming languages, it is necessary to mention the fundamental design of electronic programmable computers. Modern computers are stored program computers, i.e., machine instructions are stored in memory. This design arose in 1945 when John von Neumann, Eckhert, and Mauchly worked on the design of EDVAC (Electronic Discrete Variable Automatic Computer) [132], one of the earliest computers. The design, also known as the von Neumann architecture, allowed machines to swap instructions without altering their physical or wiring setup. Konrad Zuse invented such a design independently for his Z3 computer in 1941. The programs for the Z3 were stored on an external tape and no rewirings were necessary to change programs.

First generation programming languages (1GLs) are machine level programming languages that consist of 1s and 0s. In the case of the Z3, or Babbage's Differential Engine for example, the punched tape consists of punch holes that are physical representations of 1s or 0s. The advantage of such a language is efficiency since no intermediate compilation or assembly is necessary for the machine to execute the given instructions. However, the task of writing in 1GLs, essentially entering streams of binary numbers, is a tedious one, not to mention its error prone nature. In addition, the machine languages for different computers might be different, introducing the additional task of translating programs for execution on another computer.

Second generation languages (2GLs) are low-level assembly languages that are machine specific. They are easier to be read and written by a person, assuaging the problem of tedious writing and proneness to error. In particular, learning and error correction are easier with 2GLs. Assembly code has to be converted to binary machine code (1GLs) for execution.

Third generation languages (3GLs) are high-level programming languages that are machine independent. They include features such as named variables, assignment operators, data structures, etc. that further assuage the difficulty of writing programs. Examples of early 3GLs are COBOL [149] and Fortran [25]. Some examples of later 3GLs are C++ [166] and Java [156]. Early 3GLs are procedure-oriented languages that require programmers to specify a program in the sequence in which it is executed. This form of programming is also known as imperative programming. Later 3GLs are object-oriented such that programming tasks are arranged into objects that can be used as building blocks for larger programs.

Fourth generation languages (4GLs) differ significantly from 3GLs in that they specify what needs to be done instead of how it should be done [111]. An example is the IBM RPG (report program generator) programming language [37] that generates reports using specifications of the data and data formats involved. In this manner, they are intended to reduce, although practically not always, programming time and effort.

The fifth generation languages (5GLs) attempt to further reduce programming effort. They require programmers to specify only the problem, along with conditions or constraints that need to be satisfied. They are usually accompanied by complementary intelligent mechanisms that solve the problem automatically. An example of a fifth generation programming language is Prolog [32].

1.2 The XL Programming Language

The eXtended L-systems (XL) programming language [91] is an extension of the Java programming language. It can be seen as a 4GL based on its graph or rule-based features, i.e. how the rules are applied is transparent to the programmer. As its name suggests, an introduction to this programming language is incomplete without mentioning L-systems, a domain for which a chapter is dedicated to in this thesis. Here, beginning from the study of morphogenesis, a summarized account of historical events leading to the

development of the XL language is given.

Morphogenesis is the development of forms and patterns in living organisms. The term was coined by Johann Wolfgang von Goethe in the nineteenth century in a biological context [8]. Historically, the study of morphogenesis has been approached in two directions [138], structure-oriented, or space-oriented. The former focuses on the development of biological entities while the latter focuses on the space embedding the entities. More specifically, the structure-oriented approach describes form as a derivative of growth [169] while the space-oriented approach describes form using the locale of substances in the whole space [171]. In both approaches, it was discovered that complex patterns and morphology can result from relatively simple rules. For example, a cellular automaton [173] has rules for the state transition of cells (conforming to the space-oriented approach) while an L-system [101] has rules for rewriting strings (conforming to the structure-oriented approach).

The dawn of rule-based concepts triggered the exploration of their usage in the modelling of various biological entities such as cells, organs, and individual plants (e.g. [81, 59, 79, 164]). In 1979, Szilard and Quinton integrated concepts of geometry with plant structures generated by L-systems [167], associating the field of computer graphics with the study of morphogenesis. Many ideas then emerged in computer graphics for modelling and visualizing plants at different levels of organization (e.g [7, 162, 39, 143]).

In the early 1990s, functional-structural plant modelling emerged from the need to merge spatially-explicit representations of plants with process-based physiological models in order to take interactions between plant structure and functioning into account [95]. This domain was new and can be considered multi-disciplinary, requiring expertise in plant physiology, morphology, and for visualization, computer graphics. Early functional-structural plant models (FSPMs) focused on one aspect of physiology in relation to physical structure. For example, they considered light interception ([168, 87, 66]), assimilate allocation ([56, 17]), or xylem sapflow [60]. In most early FSPMs, causality was considered only in a uni-directional manner, from structure to functionality. In the late 1990s, FSPMs accounting for multiple aspects of plant physiology emerged ([134, 9, 178]) and some models accounting for bi-directional causality between structure and function appeared ([148, 76]).

While increasingly complex and accurate FSPMs continue to be developed [3], L-systems remain as the prevailing rule-based formalism in structural plant modelling [72]. Variants or extensions of L-systems have emerged to cope with modelling requirements - e.g., stochastic L-systems [51], context-

sensitive L-systems [153], table L-systems [86], pseudo L-systems [137], parametric L-systems [71], differential L-systems [140], growth grammars [94], environmentally-sensitive L-systems [139], and open L-systems [123].

To address some missing but desirable features of L-systems, relational growth grammars (RGG) [92] were introduced by Kniemeyer et al. Some of these features include:

- Representation of complex topologies other than trees and strings.
- An integrated representation for geometry and structures produced by rules.
- Flexibility for additional relations in addition to the conventional branching and successor relations.
- Structure-aware navigation for computations that require structural information.

Their solution was to use graphs as representation. Graphs are capable of representing complex topologies, addressing the first desired feature. Graphs can also be interpreted as scene graphs [55] that contain 3-dimensional geometrical information, striking off the second desired feature. Last but not least, edges in graphs can be labelled differently for additional relations and they allow structural navigation directly, as opposed to indirect representation of branching relations using bracket symbols in string-based L-systems. The concrete implementation of RGG, empowered with most if not all the features of the Java programming language, resulted in the XL programming language [91], the basis of extensions made in this thesis.

1.3 Computer Graphical Modelling & Level-of-Detail (LOD) Visualization

In the field of 3-dimensional (3D) computer graphics, a vast quantity of 3D data is used to render objects in a virtual scene to 2-dimensional (2D) pixels. In 1976, James Clark published a paper titled "Hierarchical Geometric Models for Visible Surface Algorithms" [30]. The paper addressed the redundancy of using many polygons to render objects covering relatively few pixels. Clark introduced several fundamental level-of-detail (LOD) techniques to reduce the redundancy, e.g. a hierarchical scene graph structure, view frustum

culling, parallel processing of scene graphs, perceptual metrics, etc. Today, the acronym *LOD* is used in several contexts, either referring to the notion of computing multiple resolutions of geometrical models, or to a particular resolution of a geometrical model. In plural, the acronym is appended with 's' to becomes *LODs*, referring to a set of resolutions of a geometrical model.

When LOD was relatively new to the graphics community, LODs were created by hand. In the 1990s, many papers were published about automating the creation while retaining visual appearance in render results. Some of these algorithms remain useful today, e.g. vertex decimation [158] and gridded vertex clustering [147]. Subsequently, other algorithms and frameworks for LOD surfaced. Some key developments include optimization-based predictive schedulers for selecting LODs [61], progressive meshes for continuous LOD [82], vertex hierarchies for view-dependent LOD [83], quadric error metrics for measuring simplification error [65], etc.. A comprehensive collection of LOD techniques can be found in the book "Level of Detail for 3D Graphics" by Luebke et al. [107]. In part II of this thesis, LOD in the context of vegetation visualization, and its relationship with relational growth grammars is addressed.

1.4 Motivations

A multiscale representation of objects in a virtual scene is necessary for the application of LOD techniques in order to achieve accelerated rendering. Such multiscale data structures are also needed in the case of plants, where different spatial and functional levels (e.g. genome, cell, tissue, organ, branch, individual plant, stand) are distinguished. A mathematical theory for a graph-based representation of such multiscale structures has been proposed by Godin and Caraglio [67]; and a corresponding relation of "refinement" or "decomposition" was foreseen as a basic edge type in XL for the purpose of connecting nodes belonging to different resolutions or scales. *However, no appropriate algorithms and functions for its use in the graph-grammar context of XL were designed and implemented, nor were any application examples tested and evaluated.*

Data representation alone is not the only reason to adopt a multiscale approach. A second motivation arises from the dynamic aspects of modelling. Considering the hierarchy of physical models (ranging from continuum models to quantum mechanics), the interest of a modeller usually resides in a pre-

dominant scale. If the scale of interest is macroscopic, the effects of inputs to the microscopic model are usually modelled by some constitutive correlations at the macro scale because representative models at the micro scale often pose computational or analytical problems. Despite their success in many applications, the extension of such correlation-based approaches to complex scenarios has proven to be difficult, often requiring complicated mathematical functions [46]. Multiscale methods have been successfully applied to overcome such difficulties [22, 175]. Eventually, several general frameworks for multiscale modelling in mathematical physics such as the heterogeneous multiscale method (HMM) by E and Engquist [47] have been developed.

1.4.1 Research Questions

In the context of the above-mentioned requirements for accelerated rendering and multi-level spatial and functional representations of plants, the following research questions are addressed in this thesis:

- Multiscale Data Structure Design
    - How can 3-dimensional structures be described consistently at several spatial scales at once, based on the graph representation used in XL?
    - How can multi-scaled graphs be transformed by rules?
    - What are the various aspects of consistency in multi-scaled graphs? How can consistency checks on these aspects be performed efficiently on multi-scaled graphs?
    - How can transformation, query and rendering of multi-scaled graphs be performed efficiently?
- Programming Language XL Extension
    - How can the programming language XL be extended for multi-scaled graphs?
    - How can classical, and, possibly new Level-of-Detail methods be incorporated in the multi-scaled graph approach of XL?
- Tests and Benchmarks

  - What are the various test data and scenarios that can considerably challenge the performance, rendering quality and integrity of multi-scaled structures?
  - Are there any unique or isolated scenarios that result in unsatisfactory performance, rendering quality or integrity of multi-scaled structures?
  - What are the differences in performance and rendering quality of multi-scaled structures as compared to existing or alternative implementations?

- Application and Usability
  - What workflows and interactive Graphical User Interface (GUI) elements are suitable for users to control the transformation, query and rendering of multi-scaled structures?

1.4.2 Research Objectives

Corresponding to the research questions, the following research objectives are achieved and specified in this thesis:

- Multi-scaled Data Structure Design
  - Establish abstract and/or theoretical concepts for representing 3-dimensional structures at several levels of spatial resolution, consistent with the graph-based approach of XL.
  - Design data structures with reference to the established concepts for multi-scaled graphs, taking into account the following:
    * Algorithms for performance and accuracy optimized queries.
    * Algorithms for verifying the consistency and integrity of the data structures across various scales.
    * Algorithms for performance and accuracy optimized transformations, particularly rule-based transformations.
    * Algorithms for parallel traversal to reap performance optimized rendering.

  - Design an export and import data format to represent the multi-scaled data structures. The data format shall be text-based and preferably XML-based [112]. The data format shall adhere as much as possible to any existing standards.

- Programming Language XL Extension
  - Extension of the programming language XL by appropriate operators, functions and supporting classes for multi-scaled data structures.

- Tests and Benchmarks
  - Perform benchmark tests for the new software components using complex scenes with vegetation models.
  - Document benchmarks and tests performed.

- Application and Usability
  - Implement established data structures, algorithms and interfaces as part of the opensource software GroIMP.
  - Improve GroIMP Graphical User Interface (GUI) elements for controlling the display, transformation and querying of multi-scaled data structures.
  - Document all software implementations and modifications.

- Evaluation and Conclusion
  - Evaluation of tests and benchmarks performed.
  - Draw conclusion for further improvements and future projects.

Part I

The Rule-Based Paradigm & XL

Chapter 2

Linear Rewriting Systems

The difficulties are such that one cannot hope to have any very embracing theory of such processes, beyond the statement of equations. It might be possible, however, to treat a few particular cases in detail with the aid of a digital computer. This method has the advantage that is is not so necessary to make simplifying assumptions as it is when doing a more theoretical type of analysis. - Alan Turing, 1952.

2.1 Introduction to Formal Languages

L-systems (short-form for Lindenmayer systems) were introduced by Aristid Lindenmayer in 1968 to model the development of multicellular organisms [101]. In the introductory chapter, they are mentioned as a type of system capable of generating complex patterns and morphologies using rewriting rules that are concise in comparison. This chapter provides some scattered glimpses of L-systems and their variants, leading towards the subject of this thesis, the XL (eXtended L-systems) programming language.

What are concise rewriting rules? We begin with an answer to this question by considering first the domain of formal languages. In a formal language, one is generally concerned with a set of finite words made up of symbols from an alphabet. The set can be specified, at least in principle, by listing its elements if it is finite. An infinite set, on the other hand, requires a finitary device for specification. Such finitary devices can be called rewriting rules, rewriting systems, grammars, automata, etc. [114]. An L-system is one such device.

2.1.1 Alphabets, Words & Languages

In this section, we describe fundamental terminology used for the description of various rewriting systems.

An *alphabet* Λ is a finite non-empty set with elements called *symbols*. A *word* over Λ is a finite sequence consisting of zero or more symbols of Λ. The word with zero symbols is called the *empty word*, denoted by λ. For example, the binary alphabet $\Lambda = \{0, 1\}$ can have words $01, \lambda, 001, 1100$ over it. The same symbol can appear multiple times in a word.

Given two words, a and b over Λ, their concatenation, ab, is obtained by writing them one after another. The set of all words over Λ and the set of all non-empty words over Λ are written as Λ^* and Λ^+ respectively. Λ^* and Λ^+ are also the *free monoid* and *semigroup* generated by Λ in algebraic terms.

A word v is a subword of a word w if there are words u_1 and u_2 such that $w = u_1 v u_2$. If u_1 is λ, v is a prefix of w. It follows that if u_2 is λ, v is a suffix of w. We call a word *primitive* if it cannot be decatenated into two or more identical words, i.e. $w = u^i$ does not hold for any word u and integer $i \geq 2$.

A *language* L over Λ is a subset, finite or infinite, of Λ^*. For example, $L_1 = \{\lambda, 1, 10, 0001\}$ and $L_2 = \{0^p | \text{p is an even integer}\}$ are languages over the binary alphabet $\Lambda = \{0, 1\}$. A finite language such as L_1 can be, at least in principle, be defined by listing all its words. This is not possible for infinite languages such as L_2. By considering languages as sets, Boolean operations like union, intersection, and complementation can be applied. The concatenation operation for words is extended for languages:

$$L_1 L_2 = \{w_1 w_2 | w_1 \in L_1 \text{ and } w_2 \in L_2\}, \text{ and}$$

$$L^n = \{w_1 w_2 ... w_n | w_i \in L, 1 \leq i \leq n\}.$$

As such, the *Kleene star* and *Kleene plus* of a language L, L^* and L^+, are defined as the union of all non-negative powers of L and all positive powers of L respectively.

Boolean operations, concatenation, and Kleene star are the regular operations for languages. Therefore, when speaking of a *regular language*, one is referring to a language L_R over Λ, where L_R can be obtained by applying the regular operations finite times on the empty language $\emptyset$ and Λ. A *star-free* language L over Λ is one that can be obtained by applying Boolean operations and concatenation finite times on $\emptyset$ and Λ without applying the Kleene

star operation.

2.1.2 The Chomsky Grammars and Hierarchy

This section gives an overview of grammars arising in classical language theory. Most of them are modifications of the classical notion of a rewriting system, introduced by Axel Thue at the beginning of the 20th century [113].

A rewriting system is a (finite) set of rules in the form $u \to v$, where u and v are words; and an occurrence of u is replaced by v each time the rule is applied.

Although rewriting systems can transform words and languages, no formal methods were available to use them defining languages. For this purpose, Chomsky introduced mechanisms in the form of grammars [27, 28], eventually being classified into the four classes of the Chomsky hierarchy of grammars and languages:

- Recursively enumerable, phrase-structure, or type 0;
- Context-sensitive, or type 1;
- Context-free, or type 2;
- Regular, or type 3.

The significance of these four classes is related to major questions in mathematics and logic. For example, type 0 grammars and langauges are, loosely said, equivalent to computability (see *Entscheidungsproblem* in 1.1.2). For a definition of Turing machines corresponding to Phrase-structure grammars, see (Def. 1.2 in [113]). Type 3 grammars and languages correspond to strictly finitary computing devices.

2.1.2.1 Phrase-structure Grammar (type 0)

A *phrase-structure* grammar is a quadruple $G = (N, T, S, P)$, where N and T are disjoint alphabets, $S \in N$, and $P \subseteq V_G^* \times V_G^*$ is a finite set of rules for $V_G = N \cup T$. N is the set of non-terminal symbols, T is the set of terminal symbols, and S is the *axiom* (the starting symbol). P is a set of rules containing ordered pairs (u, v) written in the form $u \to v$, where $u \in V_G^*$ and $v \in V_G^*$.

For $x, y \in V_G^*$, we say that x directly derives y with respect to G, $x \Longrightarrow_G y$, if and only if(iff) $x = x_1 u x_2$, $y = x_1 v x_2$, for some $x_1, x_2 \in V_G^*$ and $u \to v \in P$. The reflexive and transitive closure of the relation $\Longrightarrow$ is denoted by $\Longrightarrow^*$. The language generated by G, $L(G)$, is the set $\{x \in T^* | S \Longrightarrow^* x\}$.

Example 2.1.1 (Phrase-structure grammar). *Given type 0 grammar* $G = \{\{a\}, \{b\}, a, \{a \to a^2, a^5 \to b^5\}\}$, *the following words result from the application of the rules at each rewriting step:*

Step 0(axiom): a
Step 1: $aa = a^2$
Step 2: $a^2 a = a^3$
Step 3: $a^2 a^2 = a^4$
Step 4: $a^2 a a^2 = a^5$
Step 5: a^6 *or* b^5
.
.
.

Clearly, $L(G) = \{b^{5n} | n \geq 1\}$.

2.1.2.2 Context-sensitive Grammar (type 1)

A *context-sensitive* grammar is a type 0 grammar $G = (N, T, S, P)$ such that each rule in P is in the form $\alpha X \beta \to \alpha u \beta$, where $X \in N$, α, β, $u \in (N \cup T)^*$, and $u \neq \lambda$. In addition, P may contain the rule $S \to \lambda$ and in this case, S does not occur on the right-hand side of any rule in P.

2.1.2.3 Context-free Grammar (type 2)

A *context-free* grammar is a type 0 grammar $G = (N, T, S, P)$ such that each rule in P is in the form $u \to v$, where $u \in N$ and $v \in (N \cup T)^*$.

2.1.2.4 Regular Grammar (type 3)

A *regular* grammar is a type 0 grammar $G = (N, T, S, P)$ such that each rule in P is in the form $u \to v$, where $u \in N$ and $v \in T \cup TN \cup \{\lambda\}$.

2.2 L-systems

We now move on to the topic of L-systems. L-systems are *parallel* rewriting systems. They were originally introduced to model multicellular organisms. In the period from its introduction in 1968 to 1975, fundamental families of L-systems emerged in the L-hierarchy [86]. Parallelism is the significant characteristic of L-systems that distinguishes them from sequential grammars such as the Chomsky grammars (ref previous section).

Example 2.2.1 (Sequential vs parallel rewriting). *Given the rule $a \to a^2$ and the axiom aaa, a sequential rewriting system produces the following words at each rewriting step:*

Step 0(axiom): aaa
Step 1: a^2aa
Step 2: a^2a^2a
Step 3: $a^2a^2a^2$
.
.
Step n: a^{n+3}

That is to say, one can obtain all words a^i, $i \geq 3$ by sequential derivations of this rule. If the rewriting system is parallel, the following words are produced at each rewriting step:

Step 0(axiom): aaa
Step 1: $a^2a^2a^2$
Step 2: $a^2a^2a^2a^2a^2a^2$
Step 3: $a^4a^4a^4a^4a^4a^4$
.
.
Step n: $a^{3 \cdot 2^n}$

Only the words in the set $\{a^{3 \cdot 2^n} | i \geq 0\}$ can be obtained.

Having said that, there is an apparent correspondence of this unique characteristic with the fact that life, concerning the structure of all living things, is parallel. Consider, for example, the workings of distributed cells in an

animal. Although the variants of L-systems have not (yet) captured all intricacies of living phenomena, they have been successfully used to model and treat specific cases. The next sections provide the reader with an overview of some key variants of L-systems following the notations used in Section 2.1.

2.2.1 0L-systems & D0L-systems

0L-systems and D(eterministic)0L-systems are considered the most basic variants of L-systems. Before defining and describing these two variants, the key notion of a *finite substitution* over an alphabet Λ is given.

Definition 2.2.1 (Finite substitution). *A* finite substitution *σ is a mapping of Λ^* into the set of all finite non-empty languages which is compatible with concatenation. In other words, $\sigma(a)$ is a finite non-empty language for each $a \in \Lambda$; and $\sigma(w_1 w_2) = \sigma(w_1)\,\sigma(w_2)\,, \forall w_1, w_2 \in \Lambda^*$.*

If $\forall a \in \Lambda : \lambda \notin \sigma(a)$, we say that σ is *non-erasing* or λ-free. If $\forall a \in \Lambda : |\sigma(a)| = 1$, σ is a *homomorphism.* We can hence think of a finite substitution as a set of parallel rewriting rules in the form $u \to v$ that replace instances of symbol $u \in \Lambda$ in the string with instances of word $v \in \Lambda^*$.

Definition 2.2.2 (0L-system). *An 0L-system is a triple $G = (\Lambda, \sigma, S)$ where Λ is an alphabet, σ is a finite substitution over Λ, and $S \in \Lambda^*$ is the axiom. The language generated by G is $L(G) = \{S \cup \sigma(S) \cup \sigma^2(S) \cup ...\} = \cup_{i \geq 0} \sigma^i(S)$.*

If σ is non-erasing, the 0L-system is *propagating*, i.e. a P0L-system. The "0" in "0L-system" means that the rewriting rules are applied without (or with zero-sided) interactions between symbols being rewritten and their neighbouring symbols. Using the formal language terminology introduced earlier in section 2.1.2, this means that 0L-systems are context-free.

Example 2.2.2 (0L-system). *At this point, we take the opportunity to introduce the reader with the application of L-systems on modelling structural plant developments. Consider a 0L-system $G = (\{bud, gu\}, \sigma, bud)$ where σ is a finite substitution consisting of rules:*

$gu \to gu$,
$bud \to gu\ bud$,
$bud \to gu\ gu\ bud$.

We can also specify G by listing the range of σ*:*

$$\sigma(gu) = \{gu\},$$
$$\sigma(bud) = \{gu\ bud, gu\ gu\ bud\}.$$

It follows that $L(G) = \{gu^i\ bud | i \geq 0\}$*. We can think of this 0L-system as one that captures all possible development stages of a plant stem, each rewriting step corresponding to an elongation made of up to two growth units (gu represents a single growth unit) in one year; the non-homomorphic characteristic, i.e. two rules with the same symbol on the left-hand side, accounts for the possibility that a bud may produce one or two growth units in a year.*

Definition 2.2.3 (D0L-system). *A 0L-system is a D0L-system* $G = (\Lambda, \sigma, S)$ *iff* σ *is a homomorphism.*

Example 2.2.3 (D0L-system). *Consider a D0L-system* $G = (\{bud, gu\}, \sigma, bud)$ *where* σ *is a finite substitution consisting of rules:*

$$gu \to gu,$$
$$bud \to gu\ bud,$$

This example is similar to Example 2.2.2, except that $|\sigma(bud)| = 1$*. There is exactly one rule for each* $a \in \Lambda$*. At each rewriting step, one new word is obtained, i.e. G generates* $L(G)$ *as a sequence* $S(G)$*:*

bud
gu bud
gu gu bud
gu gu gu bud
.
.

We can think of this D0L-system as one that captures the apical development stages of a plant stem, each rewriting step corresponding to growth in one year; the homomorphic characteristic, i.e. exactly one rule with the same symbol on the left-hand side, accounts for the one and only scenario that a bud produces one growth unit a year.

2.2.2 E0L-systems

An extended 0L-system (E0L-system) [150] is a 0L-system, where the alphabet Λ is divided into two disjoint sets, non-terminals and terminals (similar to Chomsky grammars described in section 2.1.2).

The reader is encouraged to note the difference between E0L-systems from the subject of this thesis, the "eXtended L-systems (XL) programming language", although both include the term "extended". The XL programming language is capable of implementing E0L-systems but its naming is unrelated to the meaning of "extended" in E0L-systems.

Definition 2.2.4 (**E0L-system**). *An E0L-system [150] is formally defined as a 0L-system* $G = (\Lambda, \sigma, S)$ *where* $\Lambda = \Lambda_N \cup \Lambda_T$ *is an alphabet,* $\Lambda_N \cap \Lambda_T = \emptyset$*,* Λ_T *is the set of terminal symbols,* Λ_N *is the set of non-terminal symbols,* σ *is a finite substitution over* Λ*, and* $S \in \Lambda^*$ *is the axiom. In addition,* $L(G) = \cup_{i \geq 0} \sigma^i (S) \cap \Lambda_T^*$*.*

2.2.3 T0L-systems & DT0L-systems

In some cases, it may be necessary to group rewriting rules into sets so that at each rewriting step, rules from the same set are applied. These groupings are called "tables" and hence Table 0L-systems (T0L-systems) refer to 0L-systems with such groupings [77, 150].

Definition 2.2.5 (**T0L-system**). *A T0L-system is a triple* $G = (\Lambda, P, S)$ *where* P *is a finite set of finite substitutions (each a table) such that, for each* $\sigma \in P$*,* (Λ, σ, S) *is a 0L-system. The language* $L(G)$ *consists of* S *and all words in all languages* $\sigma_1 ... \sigma_k(S)$*, where* $k \geq 1$ *and* $\sigma_i \in P$*.*

If all finite substitutions in P are homomorphic, G is deterministic, i.e. a DT0L-system. Contrary to D0L-systems however (see example 2.2.3), the production of a DT0L-system is not a sequence since the order of using the tables is not included in its definition.

Example 2.2.4 (**T0L-system**). *Consider the T0L-system* $G = (\Lambda, \{\sigma_s, \sigma_w\}, ab)$*, where* $\Lambda = \{bud, gu, annualMark\}$*,* σ_s *consists of the rules:*

$bud \rightarrow gu\ bud,$
$gu \rightarrow gu,$
$annualMark \rightarrow annualMark,$

and σ_w consists of the rules:

$bud \to annualMark\ bud,$
$gu \to gu,$
$annualMark \to annualMark.$

σ_s and σ_w are two tables representing the development of a shoot in summer and winter respectively. In deciduous trees, visible markings of shoot development in winter can be observed and these are represented by the symbol $annualMark$ in G. As a result, the "summer-winter" derivations of G can be visualized as a tree diagram from top to bottom:

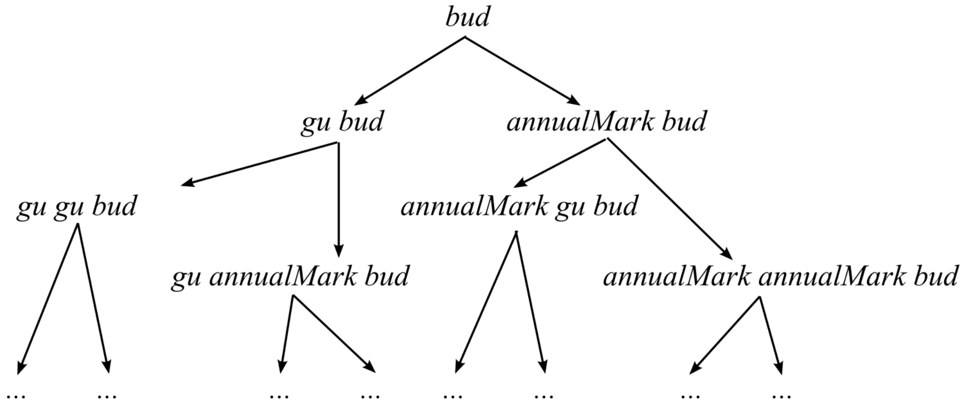


Figure 2.1: Tree diagram of T0L-system derivations

The left descendents in the tree correspond to the application of σ_s and the right descendents correspond to the application of σ_w. If the order of the table usage is given, we will be able to trace the development of the shoot spanning multiple seasons. Note that in a single summer season, σ_s can be applied multiple times and in a single winter, σ_w is usually applied only once.

2.2.4 Bracketed L-systems

Lindenmayer introduced brackets for L-systems to represent trees using strings [101]. The intention was to allow L-systems to be used as formal descriptions of branching structures found in many plants [141]. We define a bracketed D0L-system (BD0L-system) based on the definition in [142].

Definition 2.2.6 (**Bracketed D0L-system**). *Let* Λ *be a finite non-empty alphabet, the brackets* [*and*] *be two symbols outside* Λ *called* branch delimiters*. We denote the extension of* Λ *with the branch delimiters by* $\Lambda_E = \Lambda \cup \{[,]\}$*. A bracketed D0L-system (BD0L-system) is a D0L-system* $G = (\Lambda_E, \sigma, [w_0])$*, where the axiom* $[w_0]$ *is a well-nested word (for a definition of well-nested, see [142]) over the alphabet* Λ_E*, and each rule in* $\sigma \subset \Lambda_E \times \Lambda_E^*$ *has one of the following forms:*

- $a \to \alpha$*, where* $a \in \Lambda$*,* $\alpha \in \Lambda_E^*$*, and* α *is well-nested,*
- $[\to [$*, or*
- $]\to]$*.*

With a graphical interpretation of strings, bracketed L-systems can be visualized as figures resembling plants. One such interpretation is based on the notion of a LOGO-style turtle, originally introduced by Szilard and Quinton [167]. The basic idea of the turtle interpretation is that symbols or words in a given string are perceived as commands for a virtual turtle. With a set of orientation, movement, and drawing-related commands, one can specify strings that draw in a virtual space using the virtual turtle. We now briefly describe turtle interpretation in a 2-dimensional (2D) virtual space, first without the bracket symbols.

A state of the turtle is a triplet (x, y, α) where (x, y) represents the turtle's position and α represents the direction in angles which the turtle is headed towards. With a specified length d and an angle δ, the turtle can act or respond to the following commands:

F0	Move forward a step of length d. The state of the turtle changes to (x',y',α) where $x'=x + d{\cdot}\cos\alpha$ and $y'=y + d{\cdot}\sin\alpha$. The turtle draws a straight line from the point (x, y) to the point (x', y').
M0	Performs the same actions as in the command F0, but does not draw a line.
Rr	Turn right by angle δ. The state of the turtle changes to $(x,y,\alpha + \delta)$.
Rl	Turn left by angle δ. The state of the turtle changes to $(x,y,\alpha - \delta)$.

Given an unrecognized command, the turtle simply does not change its state and does not act. The image consisting of lines drawn by the turtle based on a string v of commands is called the turtle interpretation of v. Prusinkiewicz

introduced the brackets used in bracketed L-systems as commands for a turtle (see page 23 in [141]) as an extension to the original turtle interpretation:

[	Push the current state of the turtle on the stack. The information saved on the stack contains the turtle's position and orientation, as well as other attributes such as the color and width of lines being drawn.
]	Pop a state from the stack and make it the current state of the turtle. No line is drawn, although in general the position of the turtle changes.

Example 2.2.5 (**Bracketed D0L-system**). *A bracketed D0L-system* $G = (\{F, Rr, Rl, [,]\}, \sigma, F)$ *is given where* σ *consists of the following rules (for simplicity, we use* F *to represent* $F0$*):*

$$F \to F\ [\ Rr\ F\ ]\ F\ [\ Rl\ F\ ]\ F,$$
$$Rr \to Rr,$$
$$Rl \to Rl,$$
$$[\to [,$$
$$] \to].$$

G generates $L(G)$ *as a sequence* $S(G)$*:*

$$F,$$

$$F\ [\ Rr\ F\ ]\ F\ [\ Rl\ F\ ]\ F,$$

$$\begin{aligned} &F\ [\ Rr\ F\ ]\ F\ [\ Rl\ F\ ]\ F \\ &\quad [\ Rr\ F\ [\ Rr\ F\ ]\ F\ [\ Rl\ F\ ]\ F\ ] \\ &\quad F\ [\ Rr\ F\ ]\ F\ [\ Rl\ F\ ]\ F \\ &\quad [\ Rl\ F\ [\ Rr\ F\ ]\ F\ [\ Rl\ F\ ]\ F\ ] \\ &\quad F\ [\ Rr\ F\ ]\ F\ [\ Rl\ F\ ]\ F, \end{aligned}$$

$$\ldots$$

The corresponding turtle interpretation, assuming length $d = 1$ *and angle* $\delta \approx 45°$*, is shown in Figure 2.2.*

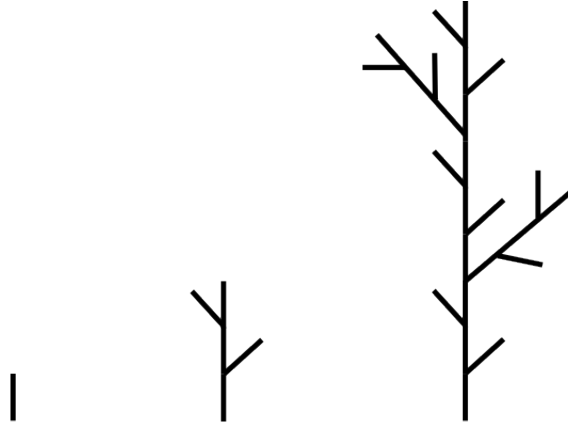


Figure 2.2: Turtle interpretation of bracketed D0L-system derivations

2.2.5 Context-sensitive L-systems

In Section 2.2.1, we saw that 0L-systems are context-free, i.e. rules are applied without regarding the context of the symbols being rewritten. In some cases it may be necessary to model interactions between neighbouring objects that are represented by neighbouring symbols in the string. Many extensions to L-systems for this purpose have been studied ([153, 77]) in the past. Two fundamental variants are 1L-systems and 2L-systems. 1L-systems are similar to 0L-systems but they consist of rules in the form:

$(* \ \alpha \ *) \ x \rightarrow \chi$, or
$x \ (* \ \beta \ *) \rightarrow \chi$

where given the alphabet Λ, α, β, $x \in \Lambda$ and $\chi \in \Lambda^*$. The context on one side, either left or right (enclosed in ($*$ and $*$)), is regarded for the application of the rule. α and β are in these cases the left and right context of x and are not rewritten. 2L-systems on the other hand consist of rules in the form:

$(* \ \alpha \ *) \ x \ (* \ \beta \ *) \rightarrow \chi$,

taking in account both the left and right context of x for the application of the rule. 0L-systems, 1L-systems, and 2L-systems all belong to a larger class of context-sensitive L-systems known as (k,l)-systems (k and l being the

length of the words constituting the left and right contexts respectively).

Example 2.2.6 (Context-sensitive L-system). *Consider the bracketed 1L-system* $G = (\{I, J, Rr, Rl, [,]\} \ , \sigma, S)$. σ *consists of the following rules (the rules that map a symbol to itself are omitted here):*

$(* \ J \ *) \ I \to J,$
$(* \ J \ [\ Rr \ *) \ I \to J,$
$(* \ J \ [\ Rl \ *) \ I \to J,$
$(* \ J \ [\ Rr \ I \ ] \ *) \ I \to J,$
$(* \ J \ [\ Rl \ I \ ] \ *) \ I \to J.$

S is the word J [Rr I] I [Rl I] I [Rr I] I. G is a 1L-system that models acropetal signal propagation, i.e. propagation of a signal from the base of a plant to its leaves. J represents a shoot segment that has already received the signal and I represents a shoot segment that has not received the signal. Notice that the first symbol in S is J, indicating that the signal begins from the base of the plant. The signal propagates to the top of the plant represented by the last symbol in 6 rewriting steps:

J [Rr I] I [Rl I] I [Rr I] I *(axiom),*
J [Rr J] I [Rl I] I [Rr I] I,
J [Rr J] J [Rl I] I [Rr I] I,
J [Rr J] J [Rl J] I [Rr I] I,
J [Rr J] J [Rl J] J [Rr I] I,
J [Rr J] J [Rl J] J [Rr J] I.
J [Rr J] J [Rl J] J [Rr J] J.

By interpreting both I and J as the turtle command $F0$, the signal propagation is visualized in Figure 2.3.

2.2.6 Stochastic L-systems

Deterministic L-systems produce languages that look artificially regular when interpreted visually. To counter such regularity, variations of the turtle interpretation, the L-system, or both by some randomization can be introduced [141]. Stochastic L-systems enable randomizations by including a probability distribution that maps the set of rules to a set of probabilities.

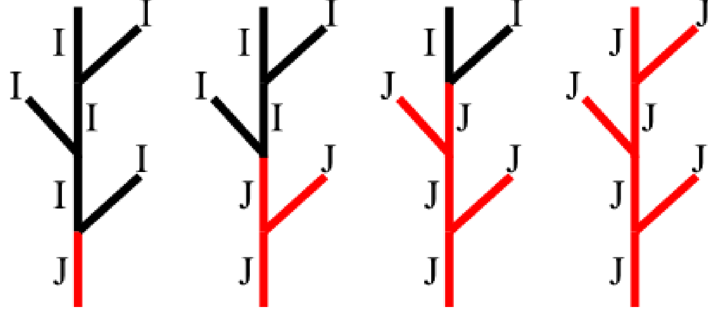


Figure 2.3: Turtle interpretation of context-sensitive 1L-system modelling acropetal signal propagation

Definition 2.2.7 (Stochastic 0L-system). *A stochastic 0L-system is a quadruplet $G = (\Lambda, \sigma, S, \pi)$ where the alphabet Λ, the finite substitution σ, and the axiom S are as defined for 0L-systems (see Section 2.2.1). Let r be a rule $a \rightarrow p$ specified in σ. π is a probability distribution function $\pi : r \mapsto (0, 1]$ such that $\sum_{p \in \sigma(a)} \pi(a \rightarrow p) = 1$, for all $a \in \Lambda$.*

In other words, the application of a rule $a \rightarrow p$ is based on the probability given by $\pi(a \rightarrow p)$. For a specific definition of stochastic 0L-systems, the reader is referred to [51].

Example 2.2.7 (Stochastic L-system). *Consider a stochastic 0L-system $G = (\{F, Rr, Rl, [,]\}, \sigma, F, \pi)$. σ consists of the following rules (the rules that map a symbol to itself are omitted here):*

$$r_1 : F \rightarrow F\ [\ Rr\ F\ ]\ F\ [\ Rl\ F\ ]\ F,$$
$$r_2 : F \rightarrow F\ [\ Rr\ F\ ]\ F\ [\ Rl\ F\ ]\ F\ [\ Rr\ F\ ]\ F.$$

π consists of the mappings $\pi(r_1) = 0.75$ and $\pi(r_2) = 0.25$, and $\sum_{p \in \sigma(F)} \pi(F \rightarrow p) = 1$. At each rewriting step, r_1 is three times more likely to be applied than r_2.

2.2.7 Pseudo L-systems

Pseudo L-systems allow substitutions from words instead of only single symbols. [137].

Example 2.2.8 (Pseudo L-system). *Consider a pseudo L-system* $G = (\{X, Y, F, Rr, Rl\}, \sigma, YF)$ *where the word* YF *is the axiom.* σ *consists of the following rules (the rules that map a symbol to itself are omitted here):*

$X\ F \rightarrow Y\ F\ Rr\ X\ F\ Rr\ Y\ F,$
$Y\ F \rightarrow X\ F\ Rl\ Y\ F\ Rl\ X\ F.$

This pseudo L-system example (taken from [137]) generates the Sierpiński "arrowhead" [108] (see Figure 2.4) by rewriting the words XF *and* YF*.*

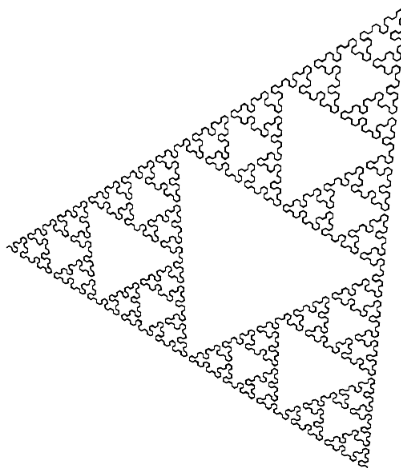


Figure 2.4: Sierpiński "arrowhead" based on turtle interpretation of pseudo L-system

2.2.8 Extending L-systems from Discrete to Continuous

2.2.8.1 Parametric L-systems

The L-systems described so far are still restricted to discrete space and time modelling. Consider for example the turtle command F that has been used in the examples (e.g. Example 2.2.5). We are able to specify unique discrete length values for the lines drawn by the turtle. To construct models that resemble biological phenomena to a greater extent, having continuous, i.e. parametric, attribute values for symbols in the string is essential. One can

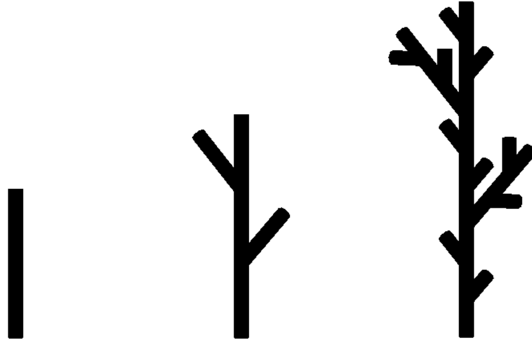


Figure 2.5: Turtle interpretation of parametric 0L-system

think of, for example, the size of cells in organisms. It is very unlikely that two cells are of the same exact size in reality.

Parametric L-systems offer a simple solution to this requirement. They consist of symbols that are appended with parameter values. For example, a parameter "length" can be introduced to the turtle command F so that the turtle moves forward for the specified distance given by the value of the parameter [94]. A parameterized symbol is called a *module*. It is easy to see that this parametric extension can be introduced to other L-system variants. A formal definition of parametric L-systems is given in [71].

Example 2.2.9 (Parametric 0L-system). *Consider a parametric 0L-system with the following rule (the rules that map a symbol to itself are omitted here):*

$$F(l) \rightarrow F(l*0.5)\ [Rr\ F(l*0.5)]\ F(l*0.5)\ [Rl\ F(l*0.5)]\ F(l*0.5).$$

At each rewriting step, the parameter l for module F is halved. The corresponding turtle interpretation is shown in Figure 2.5.

2.2.8.2 Differential L-systems

Differential L-systems (dL-systems) were introduced by Prusinkiewicz et al. in [140]. They offer the capability of relating parameters in modules (a feature introduced by parametric L-systems in Section 2.2.8.1) to ordinary differential equations (ODEs). As such, the parameter values change based on the numerical or analytical solutions of the ODEs given the differential rates, initial conditions, and boundary conditions. While the parameter values change in a continuous (time) paradigm, the structure of the L-system,

i.e. the string, is updated with these changes only at discrete time intervals at which the rules are executed. (The XL programming language has been extended to provide this feature in graph grammars by Hemmerling [75]).

2.2.9 L-systems with Global and External Interactivity

In Section 2.2.5, we described L-systems that rewrite symbols taking into account the neighbouring symbols. In [139] and [94], the authors introduced extensions to L-systems that can regard all symbols in the string, a feature known as *global* or *environmental sensitivity*.

2.2.9.1 Growth Grammars

Growth grammars [94] are extensions of parametric, stochastic 0L-systems. They are enhanced with features particularly useful for tree modelling (for application in forestry), including global sensitivity. Some of these features are:

- Representations of some 3-dimensional (3D) geometrical objects such as cylinders.
- Geometrical interpretation of 3D turtle commands such as rotations, translations, and geotropism.
- Turtle commands for modifying turtle state such as length, diameter, biomass, etc.
- Interpretive rules that operate at a second phase between a string and its structural or geometrical interpretation. The syntax for these rules are the same as the usual rules. In plant modelling, these rules can be useful to avoid cluttering the main string with symbols representing high detail (geometry) that do not contribute to developmental processes.
- Object-instancing commands to relate a previously defined sub-structure to multiple instances of a module in the string.
- Global sensitivity: *sensitive growth grammars* employ sensitive functions that operate based on the properties of other modules in the

string. For example, consider a rewriting rule that replaces a module *bud* with some *internode* modules, modelling the elongation of a shoot. The length parameter of the newly created *internode* modules can be specified with a sensitive function that relates the number of *internode*s surrounding the *bud* in 3D space to length. This requires the evaluation of the 3D positions of all *internode*s, hence the term "global sensitivity".

2.2.9.2 Environmentally-sensitive L-systems

Environmentally-sensitive L-systems [139], like growth grammars, allow rules to execute taking into account all other modules in the string. They provide a feature, called *environmental sensitivity* through special modules known as *query modules*. These modules are updated with values in the context of the environment, e.g. 3D positions and orientations during geometrical interpretation, and referenced during rule application.

2.2.9.3 Open L-systems

Growth grammars and environmentally-sensitive L-systems allow information input from the whole structure, i.e. the string operated on by the L-system. Open L-systems [123] take this sensitivity further to allow interaction with external systems. For example, consider a lattice gas automaton modelling air currents in the 3D space shared by an L-system model of a tree. The idea is that information on humidity in the earlier is required by the L-system productions in the latter, i.e. for tree growth and development; and the subsequent architectural development of the tree influences the results of the lattice gas automata model. Such bi-directional interactions of an L-system with an external system is a feature provided by open L-systems through special modules known as *communication modules*.

Chapter 3

Graph Rewriting

In chapter 2, we saw some variants of linear rewriting systems, particularly L-systems. Many of these originated from modelling certain objects or phenomena. In fact, many parts of formal language theory originated in this manner [114]. As seen in examples 2.2.2, 2.2.3 and 2.2.4, L-systems are intended and useful for models in developmental biology. Sometimes, a linear model is insufficient. In these cases, the language used to express or identify a model does not consist of strings. By using trees, graphs or multi-dimensional structures, a language becomes potentially more expressive. A comprehensive collection of non-linear formal language concepts can be found in [151].

In this chapter, we focus on graph rewriting, the most important theoretical basis of the XL programming language designed by Kniemeyer [91]. In his thesis, a wide range of graph rewriting approaches were surveyed and extensions were made for parallel graph rewriting catered primarily for functional-structural plant models (FSPMs). In the following sections, we avoid a repetition of his survey, and present instead descriptions and explanations of graph rewriting concepts that he adopted in XL. No new results are given in this chapter, but it is an important prerequisite for Chapter 4.

3.1 Introduction to Graphs

3.1.1 Fundamental Definitions

Before discussing rewriting in the context of graphs, several fundamental definitions related to graphs (based on definitions from [52] and chapter 4.1 in [91]) are given.

3.1.1.1 Alphabet

We begin with the definition of an *alphabet* that is different from that defined for linear rewriting systems in section 2.1.1.

Definition 3.1.1 (**Alphabet**). *For labelled, directed graphs, an* alphabet *is* $\Lambda = (\Lambda_V, \Lambda_E)$ *where* Λ_V *is a finite non-empty set of symbols called* node labels*, and* Λ_E *is a finite non-empty set of symbols called* edge labels.

3.1.1.2 Graph

Definition 3.1.2 (**Graph**). *Let* Λ *be an alphabet as defined in definition 3.1.1. A labelled directed* graph $G = (G_V, G_E, G_\lambda)$ *over* Λ *is a set of nodes* G_V*, a set of edges* G_E*, and a node-labelling function* $G_\lambda : G_V \longrightarrow \Lambda_V$*. The set of edges* G_E *is a subset of* $\{(s, \beta, t) | s, t \in G_V, \beta \in \Lambda_E\}$*, where* s *and* t *are called the* source node *and* target node *respectively. When referring to* G *as a mathematical set, e.g.* $a \in G$ *or* $G \subseteq H$*, we mean* $G = G_V \cup G_E$*. A* discrete graph *is a graph with* $G_E = \emptyset$.

In Definition 3.1.2, an edge is uniquely determined by its source node, target node, and edge label. Hence, parallel edges with the same edge label cannot exist. However, an edge from a node to itself, i.e. a loop is allowed.

Hereafter, we refer to the labelled, directed graph in Definition 3.1.2 whenever we use the term *graph*.

3.1.1.3 Subgraph

Definition 3.1.3 (**Subgraph**). *Let* $G = (G_V, G_E, G_\lambda)$ *be a graph over an alphabet* Λ*. A* subgraph S *of* G*, written* $S \sqsubseteq G$*, is a graph* (S_V, S_E, S_λ) *with* $S_V \subseteq G_V$*,* $S_E \subseteq G_E$*, and* $S_\lambda = G_\lambda|_{S_V}$.

3.1.1.4 Graph Homomorphism

The notion of *graph homomorphism* is repeatedly used in this thesis. We give here its definition (based on [161]) followed by an example.

Definition 3.1.4 (**Graph homomorphism**). *A (total) graph homomorphism f is a pair of functions $f = (f_V, f_E) : G \longrightarrow H$, where $G = (G_V, G_E, G_\lambda)$ and $H = (H_V, H_E, H_\lambda)$ are graphs over an alphabet Λ. f_V is a function $G_V \longrightarrow H_V$ and f_E is the function defined by $(s, \beta, t) \longmapsto (f_V(s), \beta, f_V(t))$ for all $(s, \beta, t) \in G_E$. The following two conditions must be satisfied:*

- $H_\lambda \circ f_V = G_\lambda$, *i.e.* f_V *is* label-preserving,
- $f_E(G_E) \subseteq H_E$.

For an object $a \in G_V \cup G_E$, we define $f(a) = f_V(a)$ if $a \in G_V$, otherwise $f(a) = f_E(a)$.

Example 3.1.1 (**Graph homomorphism**). *Let $G = (G_V, G_E, G_\lambda)$ be a graph over the alphabet $\Lambda = (\{bud, gu\}, \{+, >\})$ with $G_V = \{n_1, n_2, n_3, n_4, n_5\}$ and $G_E = \{(n_1, >, n_2), (n_1, +, n_3), (n_2, >, n_4), (n_3, >, n_5)\}$. The node labelling function G_λ is*

$$G_\lambda(n_i) = \begin{cases} gu, i \leq 3 \\ bud, i > 3 \end{cases}$$

where $n_i \in G_V$.

Let $L = (L_V, L_E, L_\lambda) = (\{l_1, l_2\}, \{(l_1, >, l_2)\}, L_\lambda)$ be another graph over Λ such that $L_\lambda(l_1) = gu$ and $L_\lambda(l_2) = bud$. Figure 3.1 illustrates graphs G and L.

Let m_V be a function $L_V \longrightarrow G_V$ such that $m_V(l_1) = n_2$ and $m_V(l_2) = n_4$. m_V is label-preserving since $G_\lambda \circ m_V(l_1) = L_\lambda(l_1) = gu$ and $G_\lambda \circ m_V(l_2) = L_\lambda(l_2) = bud$. m_V induces the function $m_E : L_E \longrightarrow G_V \times \Lambda_E \times G_V$, which in this case is $(l_1, >, l_2) \longmapsto (n_2, >, n_4)$. Since $(n_2, >, n_4) \in G_E$, i.e. $m_E(L_E) \subseteq G_E$, and m_V is label-preserving, m_V induces a (total) graph homomorphism $m = (m_V, m_E) : L \longrightarrow G$ shown in Figure 3.1.

Counter-example: Let c_V be a function $L_V \longrightarrow G_V$ such that $c_V(l_1) = n_1$ and $c_V(l_2) = n_2$. c_V is not label-preserving since $G_\lambda \circ c_V(l_2) \neq L_\lambda(l_2)$. c_V induces the function $c_E : L_E \longrightarrow G_V \times \Lambda_E \times G_V$, which in this case is $(l_1, >, l_2) \longmapsto (n_1, >, n_2)$. Although $(n_1, >, n_2) \in G_E$, c_V is not label-preserving and no graph homomorphism is induced by c_V. This counter-example is shown in Figure 3.2.

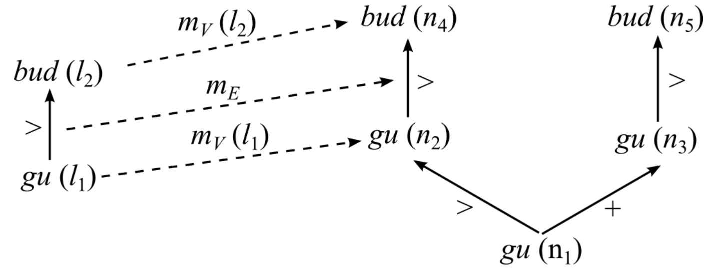


Figure 3.1: Graph homomorphism. Left: Graph L. Right: Graph G. Nodes are illustrated as text in the form "*node label*(*node*)", e.g. "*bud*(l_2)". Edges are solid arrows with their respective edge labels. Dashed arrows show the graph homomorphism $m = (m_V, m_E) : L \longrightarrow G$.

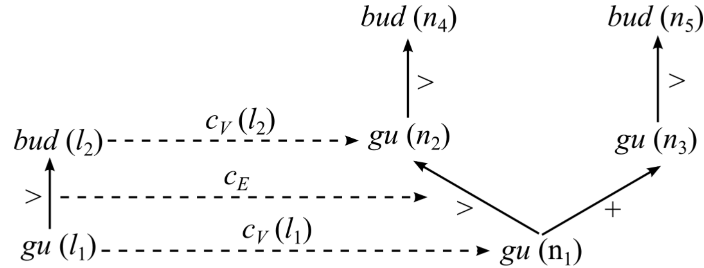


Figure 3.2: Not a graph homomorphism. Left: Graph L. Right: Graph G. Nodes are illustrated as text in the form "*node label*(*node*)", e.g. "*bud*(l_2)". Edges are solid arrows with their respective edge labels. Dashed arrows show the functions c_V and c_E. No graph homomorphism is induced by c_V since the node labels for n_2 and l_2 are different.

3.1.1.5 Partial Graph Homomorphism

Partial graph homomorphisms are used in some formal mechanisms of graph rewriting, e.g. the single-pushout approach (Section 3.2.2). We give here its definition based on [48].

Definition 3.1.5 (**Partial graph homomorphism**). *A partial graph homomorphism g is a total graph homormorphism from some subgraph $dom(g)$ of G to H, where G and H are graphs. $dom(g)$ is known as the domain of g. The injective function from $dom(g)$ to G is written $dom(g) \hookrightarrow G$.*

3.1.2 Graphs in Category Theory

Graphs and their homomorphisms form a category **Graph** in the context of category theory [49], a branch of mathematics that unifies and simplifies properties of mathematical constructions using diagrams of arrows [97].

In this section, we give the definition of *category* and descriptions of some basic gadgetries (based on [97], [6], and [161]) in category theory using examples based on graphs. These concepts are essential for the understanding of this chapter.

3.1.2.1 Category

Definition 3.1.6 (Category). *A* category Υ *consists of:*

a collection Obj of entities called objects*,*

a collection Arw of entities called arrows*,*

two assignments:

$$Arw \xrightarrow{source} Obj,$$
$$Arw \xrightarrow{target} Obj,$$

an assignment $Obj \xrightarrow{id} Arw$*, and*

a partial composition $Arw \times Arw \longrightarrow Arw$*.*

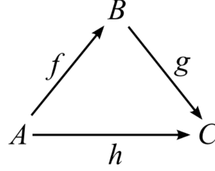
We write $A \xrightarrow{f} B$ *to indicate that* f *is an arrow with source* A *and target* B *for* $A, B \in Obj$ *and* $f \in Arw$*. The assignments* $source$ *and* $target$ *consume* f *and return* A *and* B *respectively. The notation* id_A *is used for the identity arrow assigned to each* $A \in Obj$, $A \xrightarrow{id_A} A$*. Notice that the source and target of* id_A *are both* A *and the assignment* id *consumes* A *and returns* id_A*. Certain pairs of arrows (hence* partial *composition and not* total *composition) can be composed to form another arrow. Two arrows* $A \xrightarrow{f} B_1$ *and* $B_2 \xrightarrow{g} C$ *can be composed, in that order, precisely when* B_1 *and* B_2 *are the same object to form the arrow* $A \xrightarrow{g \circ f} C$*. Category* Υ *must satisfy the following axioms:*

- *For arrows* $A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D$*, the composite arrows* $(h \circ g) \circ f$ *and* $h \circ (g \circ f)$ *must be equal, i.e. the composition of arrows is* associative*.*

- *For arrows* $A \xrightarrow{id_A} A \xrightarrow{f} B \xrightarrow{id_B} B$*,* $id_B \circ f = f = f \circ id_A$ *must hold.*

Given two objects $A, B \in Obj$ in an arbitrary category $\boldsymbol{\Upsilon}$, there may be no arrows or multiple arrows between A and B. We write $\mathrm{Hom}_{\boldsymbol{\Upsilon}}[A, B]$ to refer to the collection of all arrows between A and B in $\boldsymbol{\Upsilon}$.

Consider three arrows arranged in a triangular diagram:



The composite arrow $g \circ f$ and the arrow h gives us a parallel pair of arrows from A to C:

$$A \overset{g \circ f}{\underset{h}{\rightrightarrows}} C$$

If the two arrows are the same, i.e. $g \circ f = h$, we say that the (triangular) diagram *commutes*.

Example 3.1.2 (The category Graph). *Graphs and their homomorphisms constitute a category* ***Graph*** *where graphs are the objects and graph homomorphisms are the arrows [49].*

Let $F = (F_V, F_E, F_\lambda)$*,* $G = (G_V, G_E, G_\lambda)$*, and* $H = (H_V, H_E, H_\lambda)$ *be graphs over an alphabet* Λ*. Let* $f = (f_V, f_E) : F \longrightarrow G$ *and* $g = (g_V, g_E) : G \longrightarrow H$ *be graph homomorphisms (cf. Definition 3.1.4). In this example,* F*,* G*,* H *are objects and* f*,* g *are arrows in the category* ***Graph****. Composition of arrows, e.g. composition of* g *and* f*, is defined by* $g \circ f = (g_V \circ f_V, g_E \circ f_E)$*. The identity arrow for an object, e.g.* id_G *for* G*, is defined by* $id_G = (id_{G_V}, id_{G_E})$*, where* $id_{G_V}(a) = a$ *and* $id_{G_E}(e) = e$ *for all* $a \in G_V$ *and* $e \in G_E$*.*

3.1.2.2 Monomorphism

In a category, an arrow $B \xrightarrow{m} A$ is a *monomorphism* if for each parallel pair of arrows, $X \xrightarrow{f} B$ and $X \xrightarrow{g} B$, $m \circ f = m \circ g \Longrightarrow f = g$ [161]:

$$X \underset{g}{\overset{f}{\rightrightarrows}} B \xrightarrow{m} A$$

Example 3.1.3 (**Monomorphism in the category Graph**). *Let* $L = (L_V, L_E, L_\lambda)$ *and* $G = (G_V, G_E, G_\lambda)$ *be two graphs in the category* ***Graph*** *over the alphabet* $\Lambda = (\{bud,\ gu\},\ \{+, >\})$. *Let* m *be a homomorphism* $m = (m_V, m_E) : L \longrightarrow G$. *We illustrate* L *and* G *in Figure 3.3.* m *is a monomorphism because* L *and* G *can be conceived as sets and* m *is injective (see chapter 2 in [6] for the equivalence of injectivity and monomorphism in the category* ***Set****). In this example,* m *is not surjective and is therefore not an epimorphism (see Example 3.1.4 for epimorphism in graph categories).*

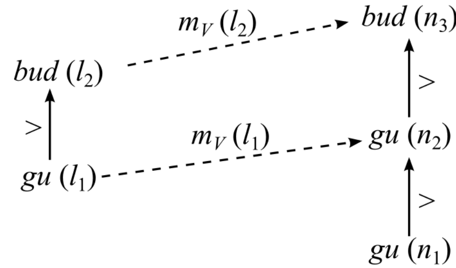


Figure 3.3: Monomorphism in the category Graph. Left: graph L. Right: graph G. m_V induces the graph monomorphism m from L to G. The induced m_E is not shown in this figure.

3.1.2.3 Epimorphism

In a category, an arrow $A \xrightarrow{e} B$ is an *epimorphism* if for each parallel pair of arrows, $B \xrightarrow{x} X$ and $B \xrightarrow{y} X$, $x \circ e = y \circ e \Longrightarrow x = y$ [161]:

$$A \xrightarrow{e} B \underset{y}{\overset{x}{\rightrightarrows}} X$$

Example 3.1.4 (**Epimorphism in the category Graph**). *Let* $L = (L_V, L_E, L_\lambda)$ *and* $G = (G_V, G_E, G_\lambda)$ *be two graphs in the category* ***Graph*** *over the alphabet* $\Lambda = (\{bud,\ gu\ \}, \{\ +, >\})$. *Let* m *be a homomorphism* $m =$

*$(m_V, m_E) : L \longrightarrow G$. We illustrate L and G in Figure 3.4. m is a an epimorphism because we treat L and G as sets and m is surjective (see chapter 2 in [6] for the equivalence of surjectivity and epimorphism in the category **Set**). In this example, m is not injective and is therefore not a monomorphism.*

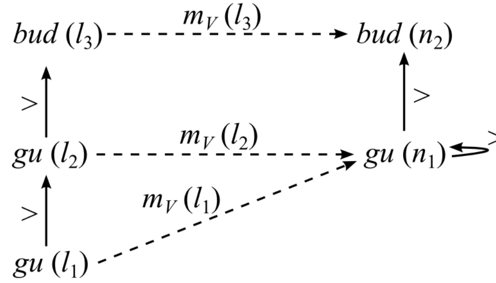


Figure 3.4: Epimorphism in the category Graph. Left: graph L. Right: graph G. m_V induces the graph epimorphism m from L to G. The induced m_E is not shown in this figure.

3.1.2.4 Isomorphism

A pair of arrows $A \xrightarrow{f} B$ and $B \xrightarrow{g} A$ such that $g \circ f = id_A$ and $f \circ g = id_B$ is an inverse pair of isomorphisms. Each arrow which has a 2-sided inverse is an *isomorphism* [161]. We say that objects A and B are *isomorphic* if an isomorphism with source A and target B, or vice versa, exists.

Example 3.1.5 (Isomorphism in the category Graph). *Let $L = (L_V, L_E, L_\lambda)$ and $G = (G_V, G_E, G_\lambda)$ be two graphs in the category **Graph** over the alphabet $\Lambda = (\{bud, gu\}, \{+, >\})$. Let m be a homomorphism $m = (m_V, m_E) : L \longrightarrow G$. We illustrate L and G in Figure 3.5. m is an isomorphism because we treat L and G as sets and m is bijective (see chapter 1 in [6] for the equivalence of bijectivity and isomorphism in the category **Set**).*

3.1.2.5 Functor

Consider a category such that the objects are categories. An arrow between objects in such a category is called a *functor* (cf. page 13 in [97]). More specifically, for objects $\boldsymbol{A}$ and $\boldsymbol{B}$, a functor $T{:}\boldsymbol{A} \longrightarrow \boldsymbol{B}$ with source $\boldsymbol{A}$ and target $\boldsymbol{B}$ consists of two related functions: the *object function* T, which assigns each object A of $\boldsymbol{A}$ to $T(A)$ of $\boldsymbol{B}$ and the *arrow function* (also written

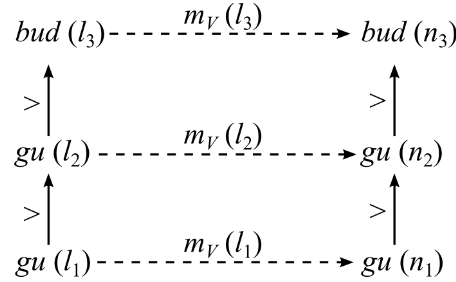


Figure 3.5: Isomorphism in the category Graph. Left: graph L. Right: graph G. m_V induces the graph isomorphism m from L to G. The induced m_E is not shown in this figure.

T), which assigns to each arrow $f : A \to A'$ of $\boldsymbol{A}$ an arrow $T(f) : T(A) \to T(A')$ of $\boldsymbol{B}$, in such a way that

$$T(id_A) = id_{T(A)}, T(g \circ f) = T(g) \circ T(f),$$

the latter whenever the composition arrow $g \circ f$ is defined in $\boldsymbol{A}$.

Example 3.1.6 (Functor). *Let* $\boldsymbol{J}$ *and* $\boldsymbol{K}$ *be two categories with a functor* T*,* $\boldsymbol{J} \xrightarrow{T} \boldsymbol{K}$*.* $\boldsymbol{J}$ *itself is a category with objects* J_0*,* J_1*, and an arrow* f*,* $J_0 \xrightarrow{f} J_1$*.* $\boldsymbol{K}$ *is a category with objects* K_0*,* K_1*...,* K_6 *connected by arrows as shown in Figure 3.6. We can view* $\boldsymbol{K}$ *as the category* ***Graph*** *such that* K_0*,* K_1*...,* K_6 *are graphs and the arrows are graph homomorphisms. The object function* T *assigns* J_0 *to* $T(J_0) = K_2$*,* J_1 *to* $T(J_1) = K_4$*, and* f *to* $T(f) = m$*, where* $K_2 \xrightarrow{m} K_4$ *is an arrow in* $\boldsymbol{K}$*.*

3.1.2.6 Category-of-Paths

We describe in this section the notion of category-of-paths (for a concrete definition, see chapter "Limits and colimits in general" in [161]). Suppose $\boldsymbol{J}$ is a category. A path through $\boldsymbol{J}$ of length $l \in \mathbb{N}$ is a list of l arrows

$$J_0 \xrightarrow{e_1} J_1 \xrightarrow{e_2} J_2 \xrightarrow{e_3} ... \xrightarrow{e_l} J_l$$

where the target of each arrow is the source of the next one, $J_0 ... J_l$ and $e_1 ... e_l$ being objects and arrows in J. A path of length 1 is a single arrow and a path of length 0 is a single object.

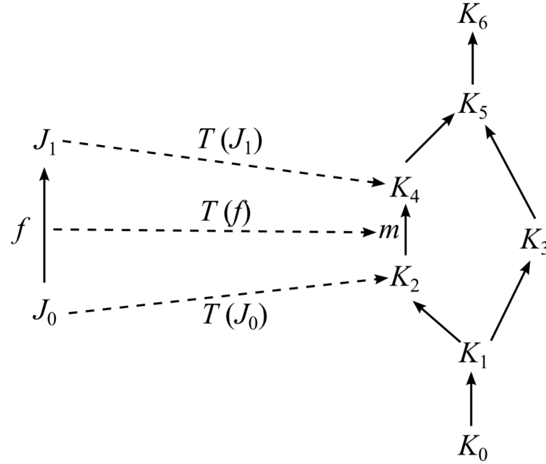


Figure 3.6: Object and arrow functions of a functor. Left: category $\boldsymbol{J}$ with objects J_0, J_1, and arrow f. Right: category $\boldsymbol{K}$ (the category $\boldsymbol{Graph}$) with objects (graphs) K_0, K_1, ... , K_6. Dashed arrows show the object and arrow functions of the functor T from $\boldsymbol{J}$ to $\boldsymbol{K}$.

We can create a category $\mathbf{Pth}(\boldsymbol{J})$, the category-of-paths through $\boldsymbol{J}$. The objects of $\mathbf{Pth}(\boldsymbol{J})$ are the objects of $\boldsymbol{J}$. The arrows of $\mathbf{Pth}(\boldsymbol{J})$ are the paths through $\boldsymbol{J}$ such that the sources and targets are the first and last objects of the respective paths. Given two arrows in $\mathbf{Pth}(\boldsymbol{J})$ (i.e. two paths of $\boldsymbol{J}$)

$$J_0 \xrightarrow{e_1} J_1 \xrightarrow{e_2} J_2 \xrightarrow{e_3} ... \xrightarrow{e_l} J_l \text{ and } J'_0 \xrightarrow{e'_1} J'_1 \xrightarrow{e'_2} J'_2 \xrightarrow{e'_3} ... \xrightarrow{e'_l} J'_l$$

with $J_l = J'_0$, the composite arrow is

$$J_0 \xrightarrow{e_1} J_1 \xrightarrow{e_2} J_2 \xrightarrow{e_3} ... \xrightarrow{e_l} J_l = J'_0 \xrightarrow{e'_1} J'_1 \xrightarrow{e'_2} J'_2 \xrightarrow{e'_3} ... \xrightarrow{e'_l} J'_l,$$

formed by ”concatenating” the paths one after another.

Example 3.1.7 (Category-of-paths). *Let $\boldsymbol{J}$ be a category with objects J_0, J_1, J_2, J_3 and arrows $J_0 \xrightarrow{e_1} J_1$, $J_0 \xrightarrow{e_2} J_2$, $J_1 \xrightarrow{e_3} J_3$. There are a total of four paths through $\boldsymbol{J}$:*

$$J_0 \xrightarrow{e_1} J_1,$$

$$J_0 \xrightarrow{e_2} J_2,$$

$$J_1 \xrightarrow{e_3} J_3, \textit{ and}$$

$$J_0 \xrightarrow{e_3 \circ e_1} J_3.$$

We construct ***Pth(J)*** *and illustrate its correspondence to* ***J*** *in Figure 3.7.*

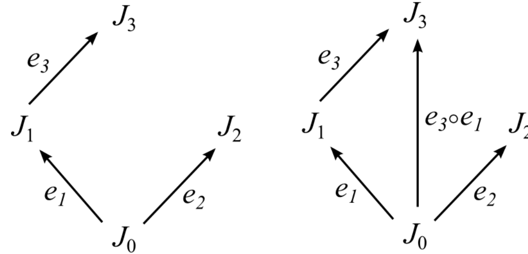


Figure 3.7: A category and its corresponding category-of-paths. Left: category $\boldsymbol{J}$. Right: category-of-paths $\mathbf{Pth}(\boldsymbol{J})$.

3.1.2.7 Diagram-in-Category

Consider a functor $T{:}\boldsymbol{J} \longrightarrow \boldsymbol{K}$ with target category $\boldsymbol{K}$. The source category $\boldsymbol{J}$ is also called the *scheme* of the functor [6]. A $\boldsymbol{J}$-*diagram-in*-$\boldsymbol{K}$ is:

a collection of objects in $\boldsymbol{K}$ $(T(J)|J$ is an object in $\boldsymbol{J})$ and

a collection of arrows in $\boldsymbol{K}$ $(T(e)|e$ is an arrow in $\boldsymbol{J})$

assigned from $\boldsymbol{J}$ [161]. The terminology follows the naming of the given categories, e.g. for a functor $D:\boldsymbol{A}\longrightarrow\boldsymbol{B}$, we say an $\boldsymbol{A}$-diagram-in-$\boldsymbol{B}$.

There is technically no difference between a scheme and a category. The alternate terminology is used to indicate a slight change of perspective from categories in general to limits and colimits, gadgetries that will be introduced in Section 3.1.2.9 [6].

Example 3.1.8 (Diagram-in-Category). *Let* $\boldsymbol{J}$ *be a category with objects* J_0, J_1, J_2, J_3 *and arrows* $J_0 \xrightarrow{e_1} J_1$, $J_0 \xrightarrow{e_2} J_2$, $J_1 \xrightarrow{e_3} J_3$*. In addition, let* $\boldsymbol{K}$ *be a category with objects* K_0, K_1, ..., K_6 *connected by arrows as shown in Figure 3.8. A functor* $T{:}\boldsymbol{J}\longrightarrow\boldsymbol{K}$ *is specified so that* $T(J_0) = K_1$,

*$T(J_1) = K_2$, $T(J_2) = K_3$, $T(J_3) = K_4$, and $T(e_1) = ek_1$, $T(e_2) = ek_2$, $T(e_3) = ek_3$. Consequently, the **J**-diagram-in-**K** consists of the objects K_1, K_2, K_3, K_4, and the arrows $K_1 \xrightarrow{ek_1} K_2$, $K_1 \xrightarrow{ek_2} K_3$, $K_2 \xrightarrow{ek_3} K_4$. We illustrate categories **J**, **K**, functor T, and the **J**-diagram-in-**K** in Figure 3.8.*

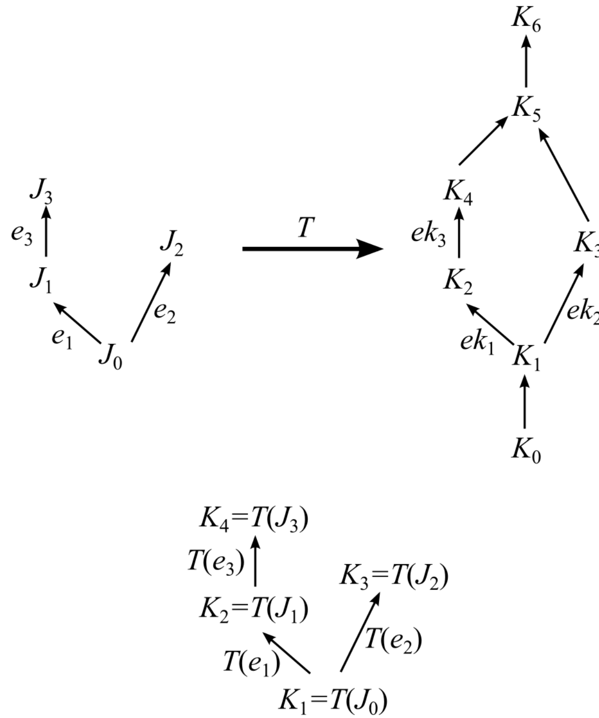
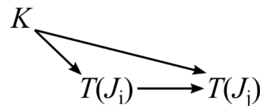


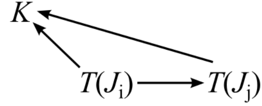
Figure 3.8: Top left: category $\boldsymbol{J}$. Top right: category $\boldsymbol{K}$. Bottom: $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$. T is a functor $T{:}\boldsymbol{J} \longrightarrow \boldsymbol{K}$.

3.1.2.8 Cone & Cocone

Consider two categories $\boldsymbol{J}$, $\boldsymbol{K}$, and a functor $T{:}\boldsymbol{J} \longrightarrow \boldsymbol{K}$. An object K in $\boldsymbol{K}$ with arrows $K \longrightarrow T(J)$ to each object $T(J)$ in the $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$ is a *cone*. For every arrow $T(J_i) \longrightarrow T(J_j)$, the triangle



must commute. A *cocone* is an object K in $\boldsymbol{K}$ with arrows $K \longleftarrow T(J)$ from each object $T(J)$ in the $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$. As in the case for cones, for every arrow $T(J_i) \longrightarrow T(J_j)$, the triangle



must commute.

Cones and cocones are defined with the aid of different supporting notions and to different extents in [97], [6], and [161]. The above description summarizes the collective meaning.

We should note that cones and cocones may or may not exist, depending on the given categories and functor. The terms *cone* and *cocone* reflect their graphical appearance when represented in a diagram, resembling geometrical cones (see Figures 3.10 and 3.11 for Examples 3.1.9 and 3.1.10 respectively). Hence the object in a cone is called the *apex* (in plural *apices*) of the cone, and the diagram-in-category accompanying the cone is called the *base* of the cone.

Example 3.1.9 (**Cone**). *Let* $\boldsymbol{J}$ *be a category with objects* J_1, J_2, J_3, *and an arrow* $J_1 \xrightarrow{e_3} J_3$. *Let* $\boldsymbol{K}$ *be a category with objects* K_0, K_1, ..., K_6 *connected by arrows as shown in Figure 3.9. We construct the categories-of-paths,* ***Pth(J)*** *and* ***Pth(K)***, *and specify a functor T:**Pth(J)**⟶**Pth(K)**.* ***J***, ***K***, ***Pth(J)***, ***Pth(K)***, *and the* ***Pth(J)****-diagram-in-****Pth(K)*** *are illustrated in Figure 3.9.* K_0 *and* K_1 *are the only objects in* ***Pth(K)*** *that have arrows to all objects in the* ***Pth(J)****-diagram-in-* ***Pth(K)***. *Taking* K_0 *as an example, a cone is formed with* K_0 *and the arrows* $K_0 \longrightarrow T(J_1)$, $K_0 \longrightarrow T(J_2)$, *and* $K_0 \longrightarrow T(J_3)$. *We illustrate this cone in Figure 3.10.* K_1 *forms a cone in the same manner as* K_0. *If we view* ***K*** *as a partially ordered set (poset), the apices of the cones formed by* K_0 *and* K_1 *are also lower bounds to* K_2, K_3, *and* K_4.

Example 3.1.10 (**Cocone**). *In this example, we reuse the categories* ***J***, ***K***, ***Pth(J)***, ***Pth(K)***, *and the* ***Pth(J)****-diagram-in-****Pth(K)*** *from Example 3.1.9. Figure 3.9 illustrates the categories and the diagram-in-category.* K_5 *and* K_6 *are the only objects in* ***Pth(K)*** *that have arrows from all objects in the* ***Pth(J)****-diagram-in-****Pth(K)***. *Taking* K_6 *as an example, a cocone is formed with* K_6 *and the arrows* $K_6 \longleftarrow T(J_1)$, $K_6 \longleftarrow T(J_2)$, *and* $K_6 \longleftarrow T(J_3)$. *We*

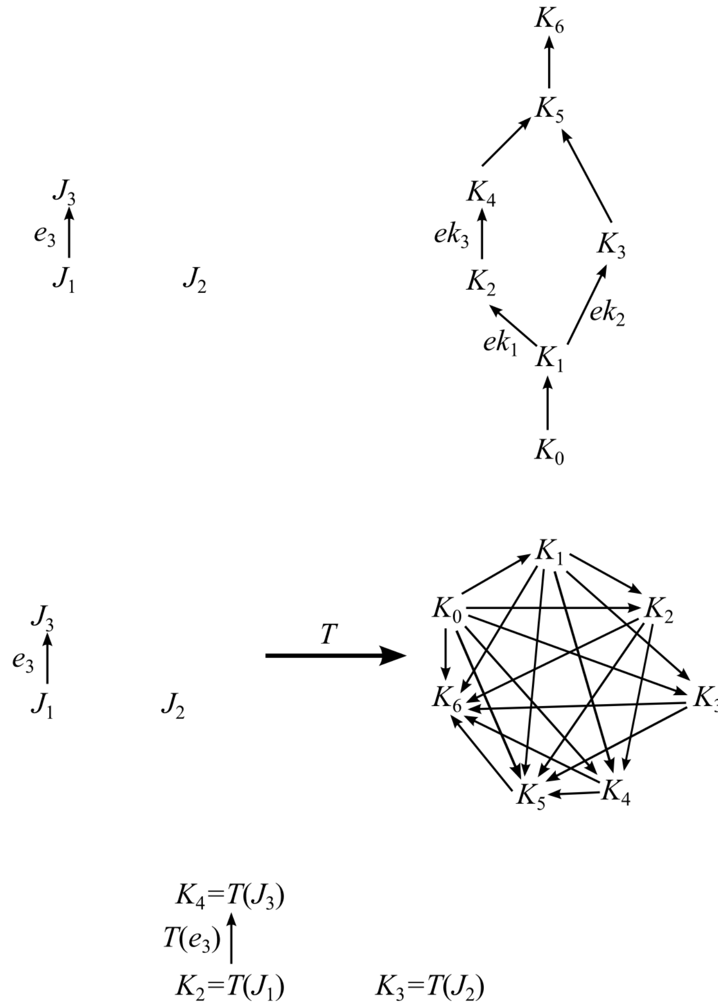


Figure 3.9: Categories forming bases of cones and cocones. Top left: category $\boldsymbol{J}$. Top right: category $\boldsymbol{K}$. Middle left: $\mathbf{Pth}(\boldsymbol{J})$. Middle right: $\mathbf{Pth}(\boldsymbol{K})$. Bottom: $\mathbf{Pth}(\boldsymbol{J})$-diagram-in-$\mathbf{Pth}(\boldsymbol{K})$. T is a functor T:$\mathbf{Pth}(\boldsymbol{J})\longrightarrow\mathbf{Pth}(\boldsymbol{K})$.

illustrate this cocone in Figure 3.11. K_5 forms a cone in the same manner as K_6. If we view $\boldsymbol{K}$ as a partially ordered set (poset), the apices of the cocones formed by K_5 and K_6 are also upper bounds to K_1, K_2, and K_3.

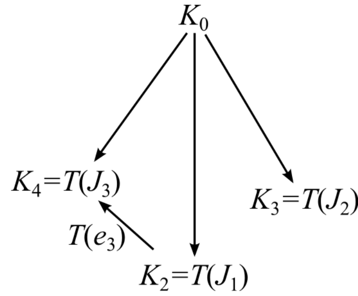


Figure 3.10: Cone. $K_2 = T(J_1)$, $K_3 = T(J_2)$, $K_4 = T(J_3)$, and $T(e_3)$ are objects and arrows in the $\mathbf{Pth}(\boldsymbol{J})$-diagram-in-$\mathbf{Pth}(\boldsymbol{K})$. K_0 is an object in $\mathbf{Pth}(\boldsymbol{K})$, in which it has arrows to K_2, K_3, and K_4, altogether forming a cone.

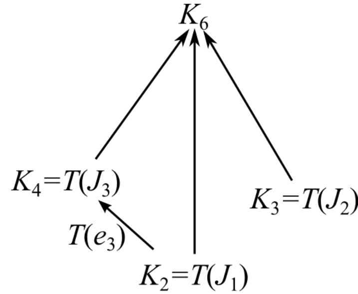
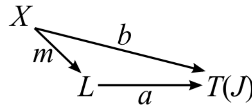


Figure 3.11: Cocone. $K_2 = T(J_1)$, $K_3 = T(J_2)$, $K_4 = T(J_3)$, and $T(e_3)$ are objects and arrows in the $\mathbf{Pth}(\boldsymbol{J})$-diagram-in-$\mathbf{Pth}(\boldsymbol{K})$. K_6 is an object in $\mathbf{Pth}(\boldsymbol{K})$, in which it has arrows from K_2, K_3, and K_4, altogether forming a cocone.

3.1.2.9 Limit & Colimit

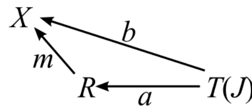
Given categories $\boldsymbol{J}$, $\boldsymbol{K}$, functor $T{:}\boldsymbol{J} \longrightarrow \boldsymbol{K}$, and the $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$, zero, one, or multiple cones (or conversely, cocones) may exist. For example, in Example 3.1.9, we saw that two cones exist in the given categories and functor.

A *limit* is a particular cone (with apex L) such that for all other cones (with apex X), there is a unique arrow $X \xrightarrow{m} L$ such that for each object J in $\boldsymbol{J}$, the triangle



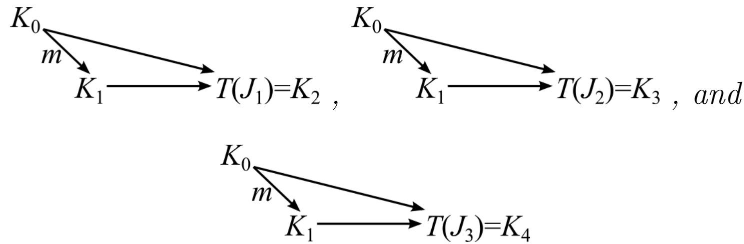
commutes [161]. Here, a belongs to the cone with apex L and b belongs to the cone with apex X. The arrow m is called the *mediator*.

Conversely, a *colimit* is a particular cocone (with apex R) such that for all other cocones (with apex X), there is a unique arrow $X \overset{m}{\longleftarrow} R$ such that for each object J in $\boldsymbol{J}$, the triangle



commutes. Here, a belongs to the cocone with apex R and b belongs to the cocone with apex X. The arrow m is also called the *mediator*.

Example 3.1.11 (**Limit**). *We continue with Example 3.1.9 and now consider both cones in the example with apices K_0 and K_1. The cones are superimposed and the mediator arrow m from K_0 to K_1 is shown in Figure 3.12. We observe that the cone with apex K_0 "factors through" the cone with apex K_1 via m. In other words, the triangles*



commute. Hence, K_1 is the apex of the limit, considering that K_0 is the apex of the only other cone in the example. If we view $\boldsymbol{K}$ *as a partially ordered set (poset), the limit with apex K_1 is also the greatest lower bound to K_2, K_3, and K_4.*

Example 3.1.12 (**Colimit**). *We continue with Example 3.1.10 and now consider both cocones in the example with apices K_5 and K_6. The cocones are superimposed and the mediator arrow m from K_5 to K_6 is shown in Figure 3.13. We observe that the cocone with apex K_6 "factors through" the*

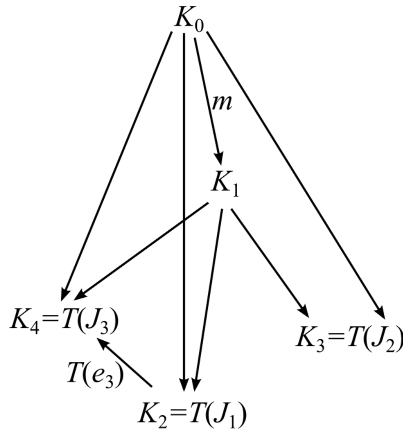
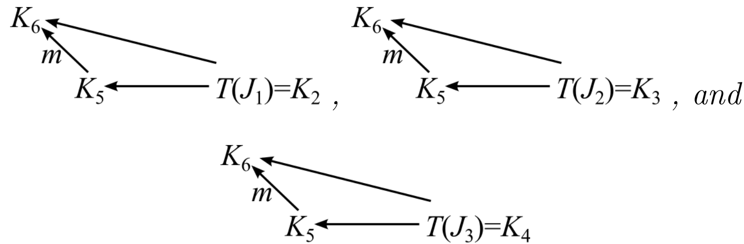


Figure 3.12: Limit. Superimposed cones with apices K_0 and K_1. $T(J_1) = K_2$, $T(J_2) = K_3$, $T(J_3) = K_4$ and the arrow $T(e_3)$ are objects and arrows that make up $\mathbf{Pth}(\boldsymbol{J})$-diagram-in-$\mathbf{Pth}(\boldsymbol{K})$, the common base for both cones. m is the mediator arrow and K_1 is the apex of the limit.

cocone with apex K_5 via m. In other words, the triangles



commute. Hence, K_5 is the apex of the colimit, considering that K_6 is the apex of the only other cocone in the example. If we view $\boldsymbol{K}$ as a partially ordered set (poset), the colimit with apex K_5 is also the least upper bound to K_2, K_3, and K_4.

Different schemes for a functor and target category result in different types of limits. Some of these types include *equalizer*, *product*, *pullback*, and conversely, *coequalizer*, *coproduct*, and *pushout*. These constructs are fundamental to graph rewriting and we describe some of them with graph-based examples in the next sections.

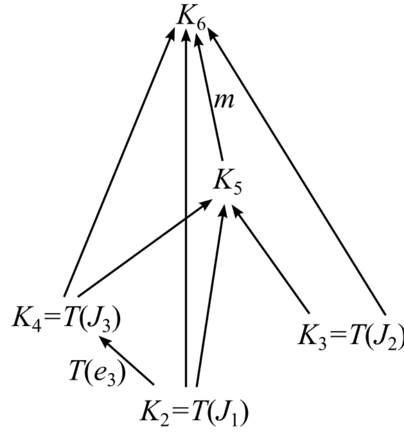


Figure 3.13: Colimit. Superimposed cones with apices K_5 and K_6. $T(J_1) = K_2$, $T(J_2) = K_3$, $T(J_3) = K_4$ and the arrow $T(e_3)$ are objects and arrows that make up $\mathbf{Pth}(\boldsymbol{J})$-diagram-in-$\mathbf{Pth}(\boldsymbol{K})$, the common base for both cocones. m is the mediator arrow and K_5 is the apex of the colimit.

3.1.2.10 Product & Coproduct

Consider a scheme $\boldsymbol{J}$, a category $\boldsymbol{K}$, a functor $T{:}\boldsymbol{J}\longrightarrow\boldsymbol{K}$, and the $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$. Given that $\boldsymbol{J}$ is a *discrete scheme*, i.e. a category with no arrows, the $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$ contains only objects and no arrows assigned by T. In such cases, the limit L of T is called the *product* and the colimit R of T is called the *coproduct* (cf. chapter 11, "Limits and colimits" in [6]).

Example 3.1.13 (Product). *Consider a discrete scheme* $\boldsymbol{J}$ *with objects* J_0*,* J_1*, and no arrows. We have a functor* $T{:}\boldsymbol{J} \longrightarrow \boldsymbol{K}$*, where* $\boldsymbol{K}$ *is the category* ***Graph*** *with objects* K_0*,* K_1*,* K_2*,* K_3*, and arrows* $K_2 \to K_0$*,* $K_2 \to K_1$*,* $K_3 \to K_0$*,* $K_3 \to K_1$*,* $K_3 \to K_2$*. The* $\boldsymbol{J}$*-diagram-in-*$\boldsymbol{K}$ *consists of* $T(J_0) = K_0$ *and* $T(J_1) = K_1$ *without arrows.* $\boldsymbol{J}$*,* $\boldsymbol{K}$*,* T*, and the* $\boldsymbol{J}$*-diagram-in-*$\boldsymbol{K}$ *are illustrated in Figure 3.14.*

We stipulate that the cone with apex K_2 *and arrows* $K_2 \to T(J_0)$ *and* $K_2 \to T(J_1)$ *is the limit, i.e. the product of* T *for any* K_3 *in the category* ***Graph****.*

K_0*,* K_1*,* K_2*,* K_3 *are graphs and the arrows in* $\boldsymbol{K}$ *are graph homomorphisms. To obtain the product* K_2 *from* K_0 *and* K_1*, we take the cartesian product of* K_0 *and* K_1*, following the formulation of products in the category* ***Set*** *(see chapter "Limits in Set" in [161]).*

Let $K_0 = (K_{0V}, K_{0E}, K_{0\lambda}) = (\{a_0, a_1\}, \{(a_0, >, a_1)\}, K_{0\lambda})$ *and* $K_1 = (K_{1V}, K_{1E}, K_{1\lambda}) = (\{b_0, b_1\}, \{b_0, >, b_1\}, K_{1\lambda})$ *be graphs over the alphabet* $\Lambda = (\{bud, gu\}, \{>, +\})$*. The labelling functions are specified as* $K_{0\lambda}(a_0) = K_{0\lambda}(a_1) = K_{1\lambda}(b_0) = gu$ *and* $K_{1\lambda}(b_1) = bud$*. Consequently,* K_2 *consists of nodes* $K_{0V} \times K_{1V}$*, edges, and labels as illustrated in Figure 3.15. The graph homomorphisms from* K_2 *to* K_0 *and* K_2 *to* K_1 *are based on projection mappings* $proj_0(K_0 \times K_1) \longrightarrow K_0$ *and* $proj_1(K_0 \times K_1) \longrightarrow K_1$ *of the nodes and edges. Figure 3.15 illustrates the graph homomorphisms in detail.*

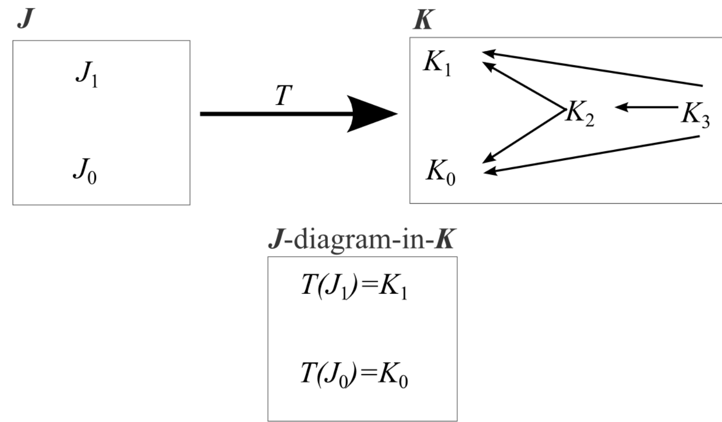


Figure 3.14: Categorical product with apex K_2. Top left: category $\boldsymbol{J}$. Top right: category $\boldsymbol{K}$. Bottom: $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$.

Example 3.1.14 (**Coproduct**)**.** *Consider a discrete scheme* $\boldsymbol{J}$ *with objects* J_0*,* J_1*, and no arrows. We have a functor* T*:*$\boldsymbol{J} \longrightarrow \boldsymbol{K}$*, where* $\boldsymbol{K}$ *is the category* ***Graph*** *with objects* K_0*,* K_1*,* K_2*,* K_3*, and arrows* $K_0 \to K_2$*,* $K_1 \to K_2$*,* $K_0 \to K_3$*,* $K_1 \to K_3$*,* $K_2 \to K_3$*. The* $\boldsymbol{J}$*-diagram-in-*$\boldsymbol{K}$ *consists of* $T(J_0) = K_0$ *and* $T(J_1) = K_1$ *without arrows.* $\boldsymbol{J}$*,* $\boldsymbol{K}$*,* T*, and the* $\boldsymbol{J}$*-diagram-in-*$\boldsymbol{K}$ *are illustrated in Figure 3.16.*

We stipulate that the cocone with apex K_2 *and arrows* $T(J_0) \to K_2$ *and* $T(J_1) \to K_2$ *is the colimit, i.e. the coproduct of* T *for any* K_3 *in the category* ***Graph****.*

K_0*,* K_1*,* K_2*,* K_3 *are graphs and the arrows in* $\boldsymbol{K}$ *are graph homomorphisms. To obtain the coproduct* K_2 *from* K_0 *and* K_1*, we take the disjoint union,* $K_0 \sqcup K_1$*, following the formulation of coproducts in the category* ***Set*** *(see chapter "Limits in Set" in [161]).*

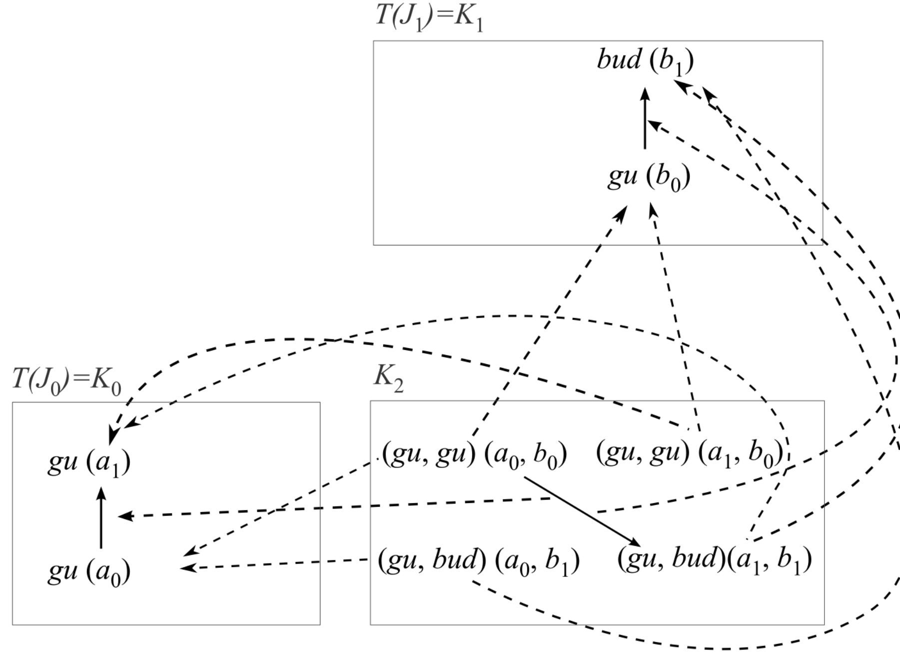


Figure 3.15: Categorical product with detailed graph homomorphisms. Top right: object (graph) K_1. Bottom left: object (graph) K_0. Bottom right: product apex K_2. Solid arrows are edges in the graphs (labels not shown). Dotted arrows are graph homomorphisms.

Let $K_0 = (K_{0V}, K_{0E}, K_{0\lambda}) = (\{a_0, a_1\}, \{a_0, >, a_1\}, K_{0\lambda})$ *and* $K_1 = (K_{1V}, K_{1E}, K_{1\lambda}) = (\{b_0, b_1, b_2, b_3, b_4\}, \{(b_0, >, b_1), (b_1, >, b_2), (b_2, +, b_3), (b_2, >, b_4)\}, K_{1\lambda})$ *be graphs over the alphabet* $\Lambda = (\{bud,\ gu\}, \{>, +\})$. *The labelling functions are specified as* $K_{0\lambda}(a_0) = K_{0\lambda}(a_1) = K_{1\lambda}(b_0) = K_{1\lambda}(b_1) = K_{1\lambda}(b_2) = gu$ *and* $K_{1\lambda}(b_3) = K_{1\lambda}(b_4) = bud$. *Consequently,* K_2 *consists of the nodes* $K_{0V} \sqcup K_{1V}$ *and edges as illustrated in Figure 3.17. The graph homomorphisms from* K_0 *to* K_2 *and* K_1 *to* K_2 *are illustrated in Figure 3.17 as well.*

3.1.2.11 Equalizer & Coequalizer

Consider a scheme $\boldsymbol{J}$, a category $\boldsymbol{K}$, a functor $T{:}\boldsymbol{J} \longrightarrow \boldsymbol{K}$, and the $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$. Let $J_0 \xrightarrow{f} J_1$ and $J_0 \xrightarrow{g} J_1$ be the objects and arrows in $\boldsymbol{J}$. The $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$ contains the objects and arrows $T(J_0) = K_0 \xrightarrow{T(f)}$

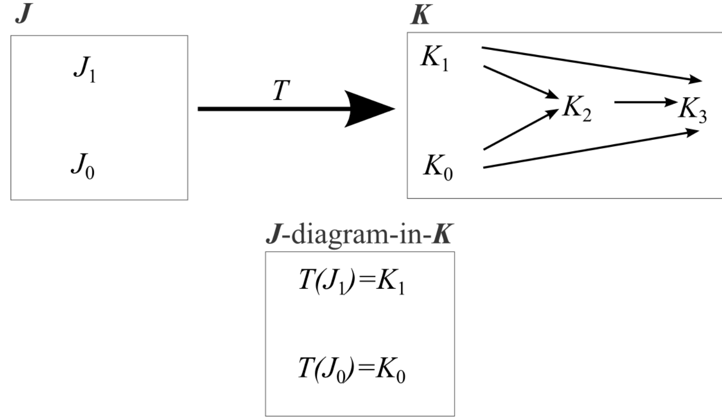
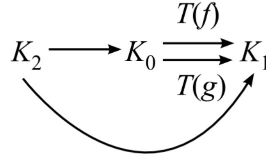


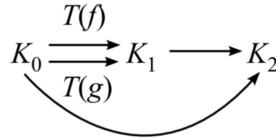
Figure 3.16: Categorical coproduct with apex K_2

$T(J_1) = K_1$ and $T(J_0) = K_0 \xrightarrow{T(g)} T(J_1) = K_1$. An *equalizer* is a limit with apex K_2 and arrows $K_2 \to K_0$ and $K_2 \to K_1$, such that $K_2 \to K_0 \xrightarrow{T(f)} K_1 = K_2 \to K_0 \xrightarrow{T(g)} K_1$ [161]:



.

Conversely, a *coequalizer* is a colimit with apex K_2 and arrows $K_1 \to K_2$ and $K_0 \to K_2$, such that $K_0 \xrightarrow{T(f)} K_1 \to K_2 = K_0 \xrightarrow{T(g)} K_1 \to K_2$:



.

Example 3.1.15 (Coequalizer). *In this example, we continue with the scheme* $\boldsymbol{J}$*, functor* T*, target category* $\boldsymbol{K}$*, and* $\boldsymbol{J}$*-diagram-in-*$\boldsymbol{K}$ *as described above in Section 3.1.2.11 for a coequalizer. In this case,* K_0*,* K_1*,* K_2 *are graphs and the arrows in* K *are graph homomorphisms.*

To derive the coequalizer apex K_2*, we view* K_1 *as a* ***Set****. Let* $\rightsquigarrow$ *be a*

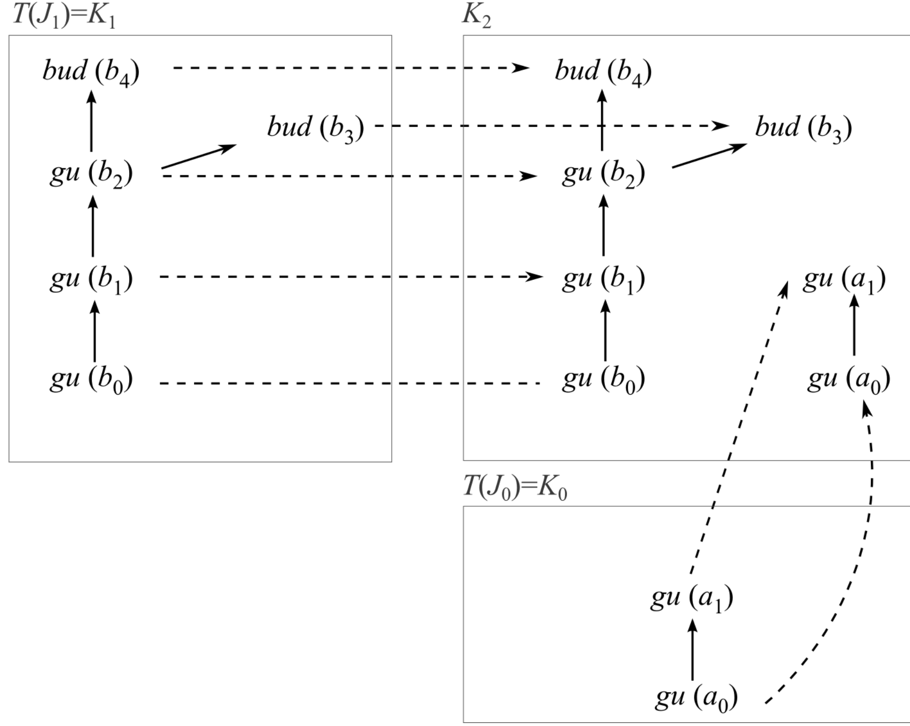


Figure 3.17: Categorical coproduct with detailed graph homomorphisms. Top left: object (graph) K_1. Top right: coproduct apex K_2. Bottom right: object (graph) K_0. Solid arrows are edges in the graphs (labels not shown). Dotted arrows are graph homomorphisms (edge mappings not shown).

relation on K_1 *such that* $k_1 \rightsquigarrow k_2 \iff \exists k_0 \in K_0 : k_1 = T \circ f(k_0)$ *and* $k_2 = T \circ g(k_0)$, $k_1, k_2 \in K_1$. *Let* $\sim$ *be the equivalence relation on* K_1 *generated by* $\rightsquigarrow$, *so that objects in* K_1 *that share a common pre-image in* K_0 *fall in the same equivalence classes. For each* $k \in K_1$, *let* $[k]$ *be the equivalence class in which* k *resides, and let* $K_1/\sim$ *be the set of all equivalence classes. Then, based on the definition of coequalizers in [161],* K_2 *is simply computed as* $K_1/\sim$.

Let $K_0 = (K_{0V}, K_{0E}, K_{0\lambda}) = (\{a_0, a1\}, \{(a_0, >, a1)\}, K_{0\lambda})$ *and* $K_1 = (K_{1V},\ K_{1E}, K_{1\lambda}) = (\{b_0, b_1, b_2, b_3\}\ ,\ \{(b_0, >, b_1), (b_2, >, b_3)\}\ , K_{1\lambda})$ *be graphs over the alphabet* $\Lambda = (\{bud,\ gu\}, \{>\})$. *The labelling functions are specified as* $K_{0\lambda}(a_0) = K_{1\lambda}(b_0) = K_{1\lambda}(b_2) = gu$, *and* $K_{0\lambda}(a_1) = K_{1\lambda}(b_1) = K_{1\lambda}(b_3) = bud$. *The function mappings* f, g, *the equivalence classes, and the derived*

coequalizer with apex K_2 are illustrated in Figure 3.18.

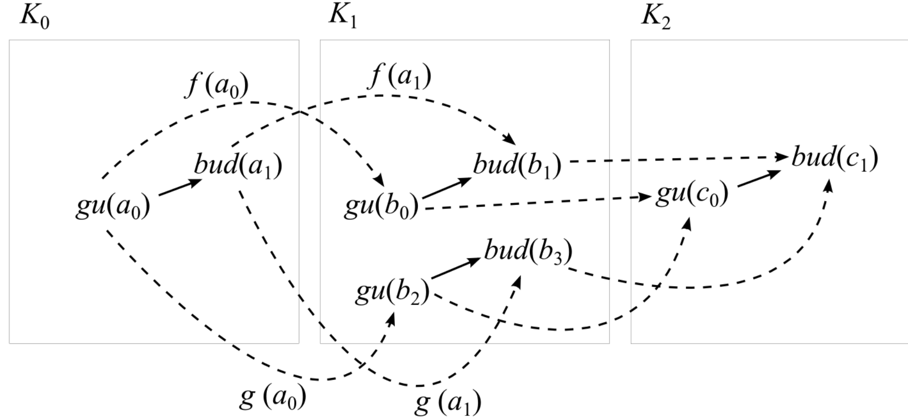


Figure 3.18: Coequalizer. Left: graph K_0. Middle: graph K_1. Right: Coequalizer apex (graph) K_2. The function mappings or homomorphisms f and g are depicted as dashed arrows from K_0 to K_1. The node c_0 represents an equivalence class containing b_0 and b_2, and c_1 represents an equivalence class containing b_1 and b_3. The equivalence classes are determined based on the equivalence relationship, i.e. objects in K_1 that share a common pre-image in K_0. Mappings for edges in the graphs are not shown.

3.1.2.12 Pullback & Pushout

Consider a scheme $\boldsymbol{J}$, a category $\boldsymbol{K}$, a functor $T{:}\boldsymbol{J} \longrightarrow \boldsymbol{K}$, and the $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$. Let $J_0 \to J_2 \leftarrow J_1$ be the objects and arrows in $\boldsymbol{J}$, and $T(J_0) = K_0 \to T(J_2) = K_2 \leftarrow T(J_1) = K_1$ be the objects and arrows in the $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$. A *pullback* is a limit with apex K_3 and arrows $K_3 \to K_0$ and $K_3 \to K_1$, such that the square

$$\begin{array}{ccc} K_2 & \longleftarrow & K_1 \\ \uparrow & & \uparrow \\ K_0 & \longleftarrow & K_3 \end{array}$$

commutes.

Conversely, let $J_0 \leftarrow J_2 \rightarrow J_1$ be the objects and arrows in $\boldsymbol{J}$, and $T(J_0) = K_0 \leftarrow T(J_2) = K_2 \rightarrow T(J_1) = K_1$ be the objects and arrows in the $\boldsymbol{J}$-diagram-in-$\boldsymbol{K}$. A *pushout* is a colimit with apex K_3 and arrows $K_3 \leftarrow K_0$ and $K_3 \leftarrow K_1$, such that the square

$$\begin{array}{ccc} K_2 & \longrightarrow & K_1 \\ \downarrow & & \downarrow \\ K_0 & \longrightarrow & K_3 \end{array}$$

commutes.

Pushouts are fundamental to graph rewriting. They always exist in the category ***Set*** [97] and the category ***Graph*** (cf. page 50 in [91]). Next, we look at an example of a pushout in the category ***Graph***. This serves as a lead in to the next sections on rewriting.

Example 3.1.16 (Pushout). *We continue in this example from the above-mentioned set up of categories for a pushout. Let* $K_0 = (K_{0V}, K_{0E}, K_{0\lambda}) = (\{a_0, a1\}, \{(a_0, >, a1)\}, K_{0\lambda})$, $K_2 = (K_{2V}, K_{2E}, K_{2\lambda}) = (\{c_0, c1\}, \{(c_0, >, c1)\}, K_{2\lambda})$*, and* $K_1 = (K_{1V}, K_{1E}, K_{1\lambda}) = (\{b_0, b_1, b_2, b_3, b_4\}, \{(b_0, >, b_1), (b_1, >, b_2), (b_2, +, b_3), (b_2, >, b_4)\}, K_{1\lambda})$ *be graphs over the alphabet* $\Lambda = (\{bud, gu\}, \{>, +\})$*. The labelling functions are specified as* $K_{0\lambda}(a_0) = K_{0\lambda}(a_1) = K_{1\lambda}(b_0) = K_{1\lambda}(b_1) = K_{1\lambda}(b_2) = K_{2\lambda}(c_0) = K_{2\lambda}(c_1) = gu$ *and* $K_{1\lambda}(b_3) = K_{1\lambda}(b_4) = bud$*.*

To obtain the apex of the pushout, K_3*, the disjoint union of the nodes in* K_{0V} *and* K_{1V} *is first computed, resulting in* $K_{0V} \sqcup K_{1V}$*. Let* f *be* $K_2 \rightarrow K_0$ *and* g *be* $K_2 \rightarrow K_1$*. A relation* $\rightsquigarrow$ *on* $K_{0V} \sqcup K_{1V}$ *is such that* $k_1 \rightsquigarrow k_2 \iff \exists k_0 \in K_{2V} : k_1 = f(k_0)$ *and* $k_2 = g(k_0)$, $k_1, k_2 \in K_{0V} \sqcup K_{1V}$*. Let* $\sim^*$ *be the (least) equivalence relation on* $K_{0V} \sqcup K_{1V}$ *generated by* $\rightsquigarrow$*, so that objects in* $K_{0V} \sqcup K_{1V}$ *that share a common pre-image in* K_2 *fall in the same equivalence classes. For each* $k \in K_{0V} \sqcup K_{1V}$*, let* $[k]$ *be the equivalence class in which* k *resides, and let* $K_{0V} \sqcup K_{1V}/\sim^*$ *be the set of all equivalence classes. Then the set of nodes* K_{3V} *in the pushout apex is* $K_{0V} \sqcup K_{1V}/\sim^*$*, and the edges* K_{3E} *are uniquely determined by the nodes. The labelling function* $K_{3\lambda}$ *follows the labelling in* K_0 *and* K_1*. The detailed graph homomorphisms are illustrated in Figure 3.19.*

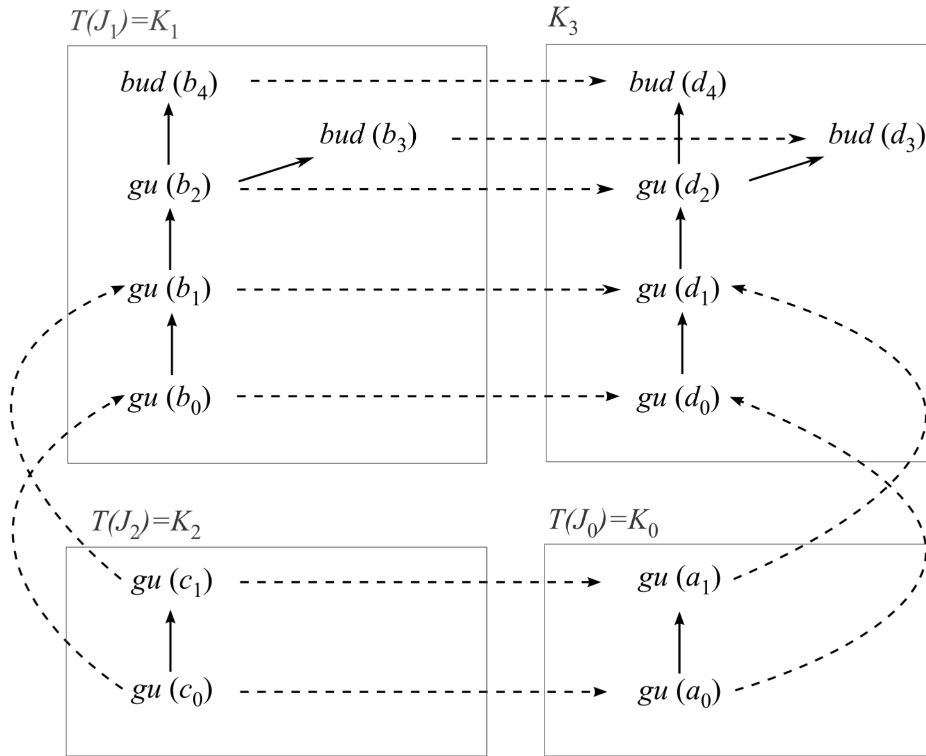


Figure 3.19: Categorical pushout with detailed graph homomorphisms. Top left: object (graph) K_1. Top right: pushout apex (graph) K_3. Bottom left: object (graph) K_2. Bottom right: object (graph) K_0. Solid arrows are edges in the graphs (labels not shown). Dotted arrows are graph homomorphisms (edge mappings not shown). The node d_0 represents an equivalence class containing a_0 and b_0, and the node d_1 represents an equivalence class containing a_1 and b_1. The rest of the nodes, i.e. d_2, d_3 and d_4 are equivalence classes on their own.

3.2 Fundamentals of Graph Rewriting

With the core definitions in place, we begin to describe graph rewriting. Graph rewriting or graph grammars provide a mechanism to model graph transformations in a mathematically precise way [52].

As in the case of rules in the form $u \to v$ for Chomsky grammars, we have, analogously, *productions* in the form $p : L \rightsquigarrow R$ for graph rewriting,

where p is the production, L is the *left-hand side* graph, and R is the *right-hand side* graph [36]. Given a graph G, an occurrence of L in G, called a *match*(m), is replaced by R when p is applied. Such an application of p for a match m is called a *direct derivation*, denoted $G \overset{p,m}{\Longrightarrow} H$, where H is the resulting *derived graph* from the replacement. A *match* is formally defined as a graph homomorphism (see Definition 3.1.4).

A direct derivation $G \overset{p,m}{\Longrightarrow} H$ generally consists of three steps. We illustrate these three steps that result in H with reference to Figure 3.20:

- Step 1: Each object (node or edge) of L is mapped to a corresponding object in G, such that the graphical structure and the labels are preserved. Between L and R, corresponding objects are also identified. The correspondences are shown using the symbols $*$, $**$, and $\#$ in Figure 3.20.

- Step 2: Every object (node or edge) in G with a corresponding object in L but no corresponding object in R, i.e. bud^{**} and the edge marked with $^{\#}$, is deleted.

- Step 3: Every object (node or edge) in R without a corresponding object in L, i.e. all objects in R except gu^{*}, is added to G.

Of course, this general description of graph rewriting is not precise and does not account for all cases. In the next sections, several formal methods for sequential (not parallel) graph rewriting are specified.

Gluing approaches to graph rewriting are generalizations of string concatenation in linear rewriting systems to gluing construction in graphs. These approaches are also commonly known as algebraic approaches [36] because they are usually defined using the *pushout* algebraic construct (see Section 3.1.2.12). In this thesis, two fundamental gluing approaches related to XL are described. They are the *double-pushout* (DPO) approach [49] and the *single-pushout* approach (SPO) [106].

3.2.1 The Double-Pushout Approach (DPO)

The *double-pushout* approach [49, 36] represents a production $p : L \rightsquigarrow R$ as a pair of total graph homomorphisms $L \overset{l}{\longleftarrow} K \overset{r}{\longrightarrow} R$. K is an *interface graph* that consists of the nodes and edges common between L and R, i.e. nodes and edges that are so-called "the same" in L and R.

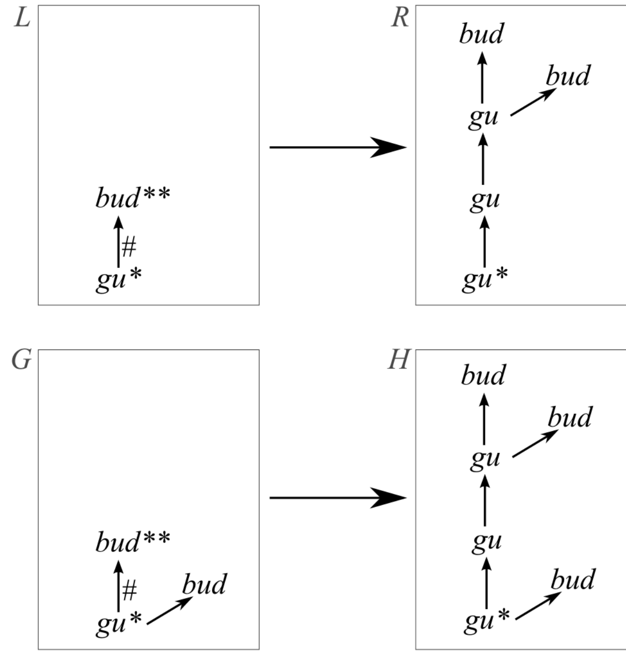


Figure 3.20: Graph rewrite. Top left: left-hand side graph L of production. Top right: right-hand side graph R of production. Bottom left: the original graph G. Bottom right: derived graph H after a direct derivation. The symbols *, ** and $^\#$ mark corresponding nodes and edges in the graphs.

A direct derivation using the double-pushout approach is modelled as two pushout (square) diagrams of graphs and total graph homomorphisms, (1) and (2) as seen below:

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ m\downarrow & (1) & \downarrow & (2) & \downarrow m^* \\ G & \xleftarrow[l^*]{} & D & \xrightarrow[r^*]{} & H \end{array}$$

. For a given graph G and a match m, the *context graph* D consists of the nodes and edges in G *except* those with a homomorphic pre-image in L and no subsequent homomorphic pre-image in K. In other words, $D = G - \{m(a)|a \in L, l(b) \neq a, \forall b \in K\}$. H is then obtained as $(D \sqcup R)/\sim^*$ (see Example 3.1.16 for details of obtaining the apex of a pushout). In addition,

the match m must satisfy certain conditions collectively known as the *gluing condition*, which we will describe after the following example.

Example 3.2.1 (Double-Pushout). *Consider a production $p : L \rightsquigarrow R$, a graph G, and a match m as shown in Figure 3.21. m is the graph homomorphism shown by the dashed arrows from L to G. K consists of the gu node identified to be the same node in L and R. To obtain the context graph D, the node bud* and the edge marked with $^{\#}$ are removed from G. H is obtained as $(D \sqcup R)/\sim^*$ to form the square pushout diagram containing K, R, D, H and the graph homomorphisms between them.*

The gluing condition consists of an *identification condition* and a *dangling condition*. Rewriting via the DPO approach is not allowed when either of these conditions fail. To illustrate the *identification condition*, consider a modified version of Example 3.2.1 shown in Figure 3.22. The left-hand side graph L now consists of an additional *bud* node that is mapped to *bud** in G. The additional bud node is not deleted by an application of the production. Hence, a conflict arises because *bud** in G is mapped from a node that is deleted by the production and also from a node that is retained in the production. Therefore, this illustrated match fails the identification condition, which requires every element of G that should be deleted by the production to have only one homomorphic pre-image in L. In Figure 3.22, *bud** in G has two pre-images in L.

To illustrate the *dangling condition*, consider another modified version of Example 3.2.1 shown in Figure 3.23. L consists of an additional *bud* node matched to the unmarked *bud* node in G. The production specifies that this *bud* node should be deleted when a derivation occurs, resulting in a dangling edge in the context graph D. This is erroneous because by Definition 3.1.2, edges must have a source and a target node.

3.2.2 The Single-Pushout Approach (SPO)

In the *single-pushout* approach [48], a production p is represented as an injective partial graph homomorphism (Section 3.1.5) r from a left-hand side graph L to a right-hand side graph R. In short, we write $p : (L \xrightarrow{r} R)$. As in the case for DPO, an occurence of L in a given graph G is a match m. The partial homomorphism r specifies which objects (nodes and edges) in L correspond to which in R. Objects mapped by r are analogous to the objects of the interface graph in the DPO approach. A direct derivation $G \overset{p,m}{\Longrightarrow} H$

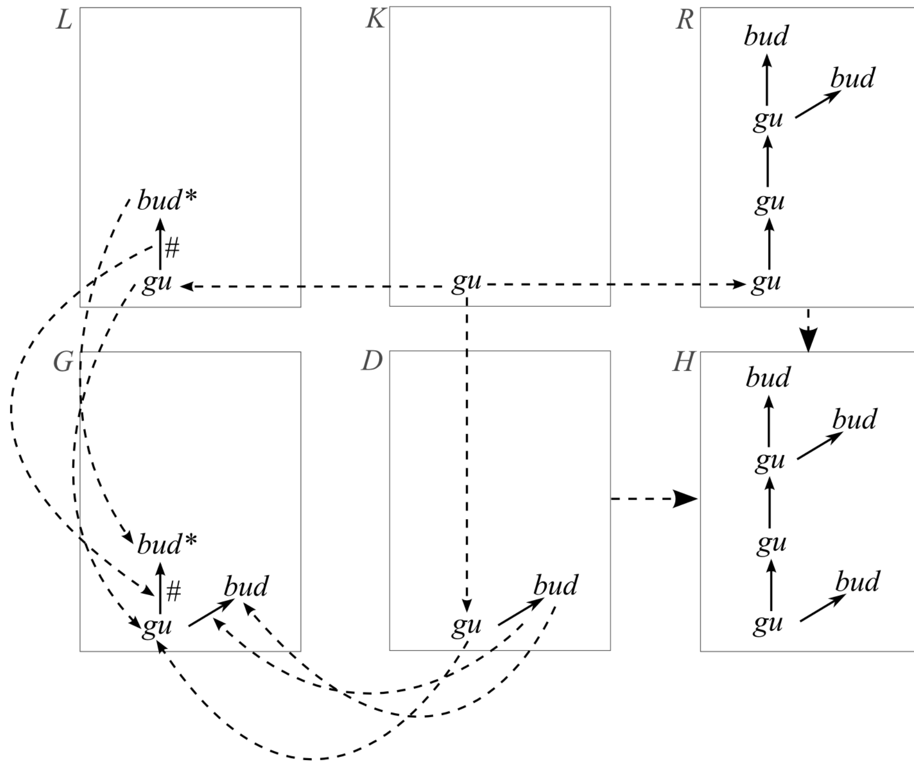


Figure 3.21: DPO graph rewrite. Top left: left-hand side graph L of production. Top center: interface graph K. Top right: right-hand side graph R of production. Bottom left: the original graph G. Bottom center: context graph D. Bottom right: derived graph H after a direct derivation. Solid arrows show edges in the respective graphs. Dashed arrows show graph homomorphisms. The detailed mappings for the co-match $m^* : R \longrightarrow H$ and $r^* : D \longrightarrow H$ are collectively represented by the larger dashed arrows to avoid cluttering.

using the SPO approach is the construction of a pushout in the category $\textit{\textbf{Graph}}^{\mathbf{P}}$, in which graphs are objects and partial graph homomorphisms are arrows. In the following example, we illustrate the details of such a construction.

Example 3.2.2 (**Single-Pushout**). *Consider a production $p : (L \xrightarrow{r} R)$ and a graph G as shown in Figure 3.24. There is a match m depicted as the total graph homomorphism from L to G. H is yet unknown and it will*

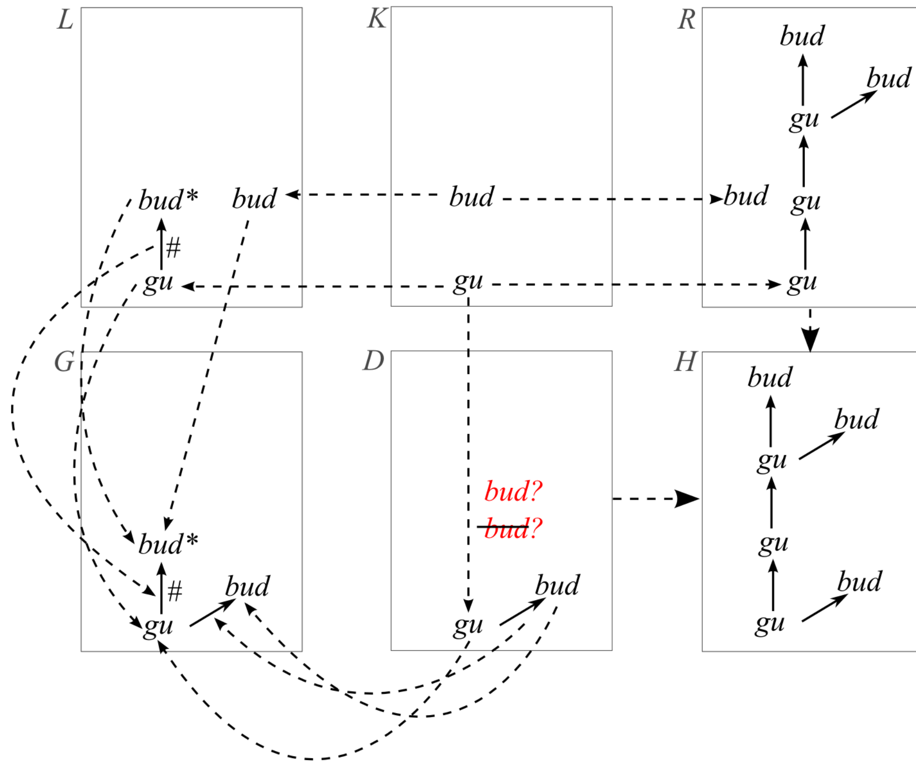


Figure 3.22: DPO identification condition failure. It is ambiguous whether *bud** should be deleted or retained in the context graph D when the identification condition is not satisfied.

*be the result of a direct derivation via the construction of a pushout in the category **Graph**$^{\mathbf{P}}$. This pushout is the result of two steps (see [48] for proof and details):*

- *Gluing: From r, we obtain the subgraph $dom(r)$ of L, which has total graph homomorphisms to R and G (see Figure 3.25). A graph D is constructed as the result of a pushout in the category **Graph** using $dom(r) \to R$ and $dom(r) \to G$.*

- *Deletion: H is obtained via the construction of a co-equalizer of the partial homomorphisms $L \to R \to D$ and $L \to G \to D$ in the category **Graph**$^{\mathbf{P}}$ (see Figure 3.26).*

An overview of the complete SPO approach is shown in Figure 3.27.

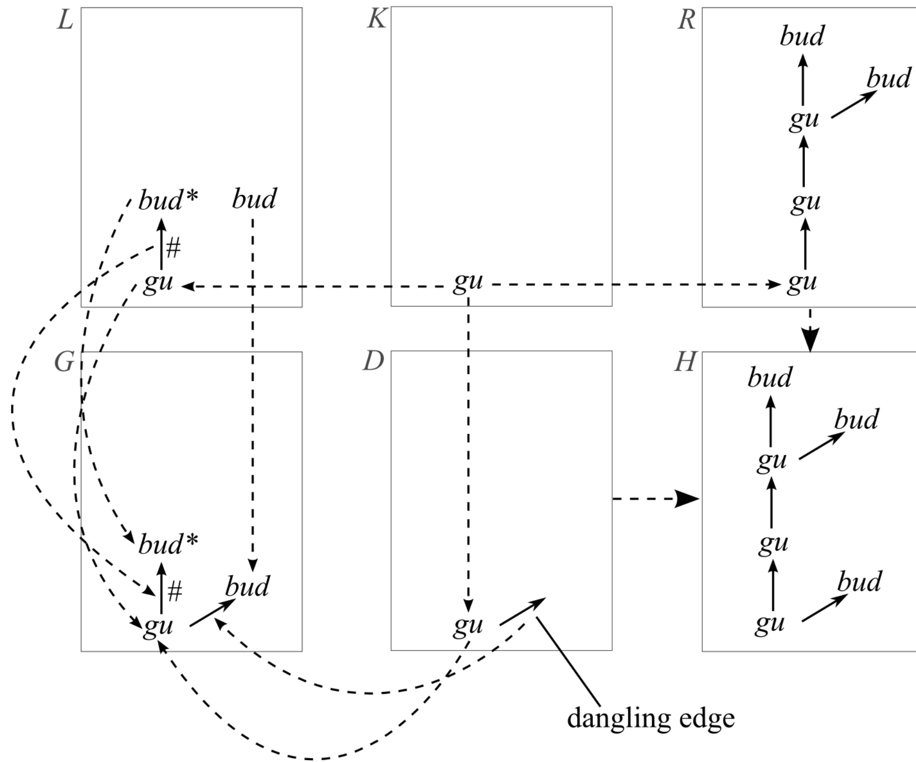


Figure 3.23: DPO dangling condition failure. The deletion of the unmarked *bud* node in G leaves a dangling edge in D.

Because of the order in which H is derived, deletion takes priority over preservation and dangling edges are removed automatically. Hence, the SPO approach overcomes conflicting and erroneous scenarios such as those depicted in Figures 3.22 and 3.23 by deleting the conflicting nodes and the dangling edges respectively.

3.2.3 Neighbourhood Controlled Embedding

A constrasting type of graph rewriting approach is the *connecting* approach. These approaches do not rely on the identification of common objects between the left-hand side and right-hand side of a rule like the gluing approaches. They explicitly define what edges to establish between the remaining host graph G^- and the graph R on the right-hand side of a rule, whenever a match

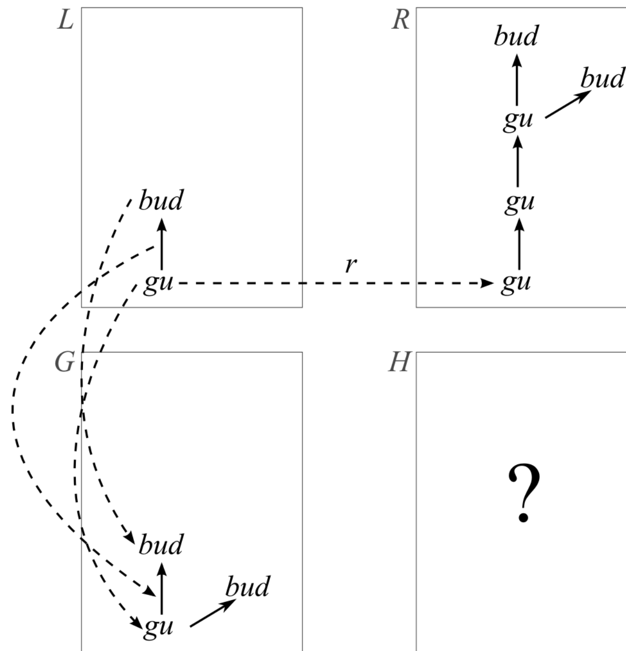


Figure 3.24: SPO production and match. Top left: left-hand side graph L of production. Top right: right-hand side graph R of production. Bottom left: the original graph G. Bottom right: unknown derived graph H. Solid arrows show edges in the respective graphs. Dashed arrows show graph homomorphism mappings. r is a partial graph homomorphism. The total graph homomorphism from L to G represents a match.

m is removed from graph G. One such approach is *neighbourhood controlled embedding* (NCE) [52].

A simple version of NCE is the *node label controlled* (NLC) mechanism for node-labelled, undirected graphs. We illustrate this technique here based on the description by Kniemeyer [91].

In NLC, a rule has the form $L \xrightarrow{e} R$, where L is one node, R is a graph, and e is a set of *connection instructions*. Each connection instruction is a pair of node labels (μ, ν), where μ and ν are node labels. For a match m, the matched node is removed from the host graph G along with its incident edges, and an isomorphic copy of R, R^c, is connected to the remaining host graph G^-. For each neighbour a of the removed node (before it was removed) in G^-, and every node b in R^c, an edge is established if the set of connection

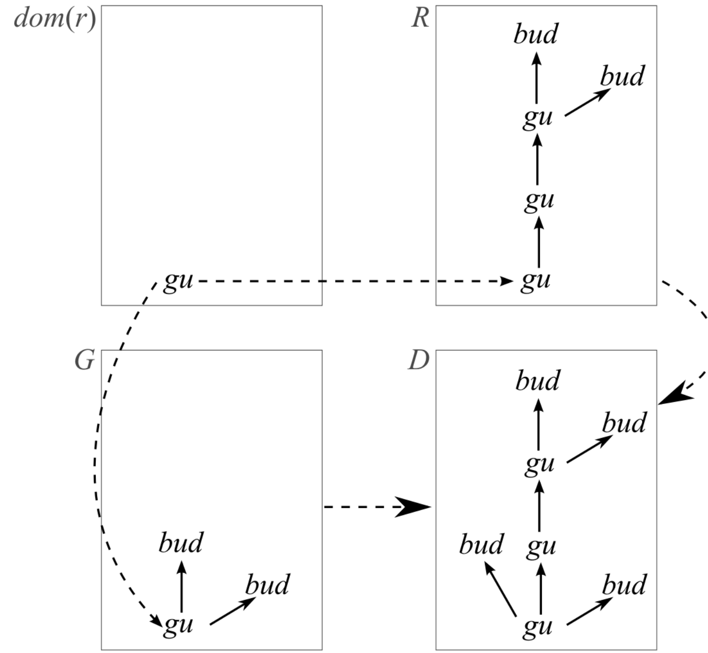


Figure 3.25: SPO step 1 - gluing. Top left: subgraph $dom(r)$ of L (L in Figure 3.24). Top right: right-hand side graph R of production. Bottom left: the original graph G. Bottom right: graph D constructed as the result of a pushout in category ***Graph***. Solid arrows show edges in the respective graphs. Dashed arrows show total graph homomorphism mappings. The graph homomorphisms $R \to D$ and $G \to D$ are collectively represented as the larger dashed arrows to avoid cluttering.

instructions contains the pair $(G_\lambda(a), R^c_\lambda(b))$.

An elaborated version of NLC is the *edge-labelled directed neighbourhood controlled embedding* (edNCE) mechanism (cf. [52] and page 46 in [91]). A production p in edNCE is in the form $L \xrightarrow{e} R$, where L and R are graphs, and e is a set of connection instructions. Each connection instruction is a tuple $(\mu, \nu, \gamma/\delta, w, d) \in L_V \times \Lambda_V \times \Lambda_E \times \Lambda_E \times R_V \times \{in, out\}$.

edNCE operates in the following manner: μ is a node in L. In a match m where $m(L)$ is removed from host graph G, nodes labelled ν in G^- with an edge labelled γ to $m(\mu)$ are identified. Edges labelled δ are established from these identified nodes to the isomorphic copy of w. d is a flag indicating the directions of the edges.

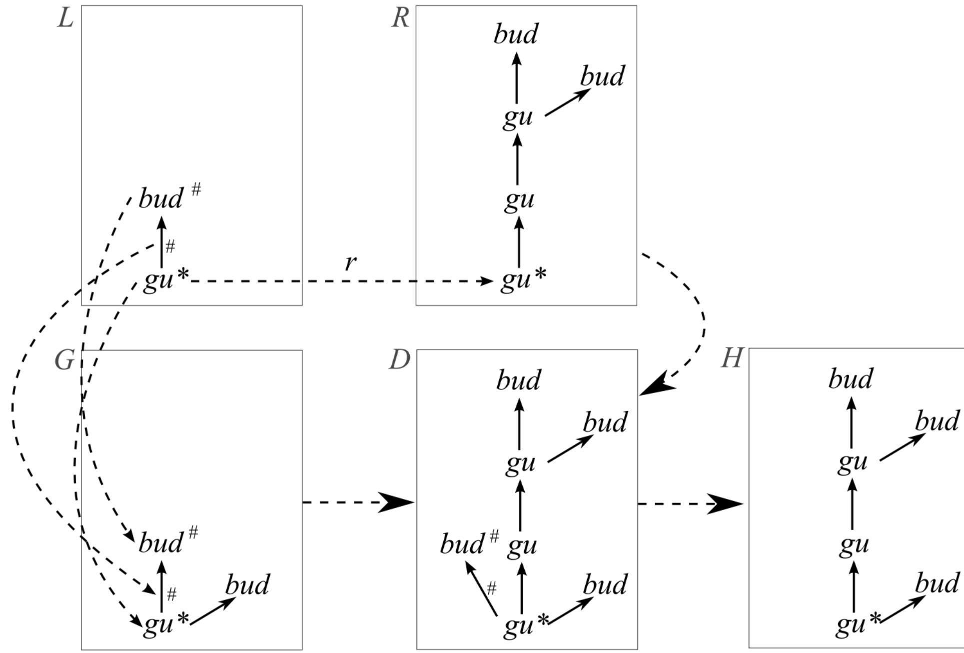


Figure 3.26: SPO step 2 - deletion. H is the result of a co-equalization of the partial homomorphisms $L \to R \to D$ and $L \to G \to D$. Let $L \to R \to D = a$ and $L \to G \to D = b$. Then the co-equalizer is H and $D \to H$, where $H \subseteq D$ is the largest subgraph of $[a(L) \cap b(L)] \cup [\overline{a(L)} \cap \overline{b(L)}]$. Here, $[a(L) \cap b(L)] = gu^*$, and $[\overline{a(L)} \cap \overline{b(L)}]$ consists of all objects in D except gu^*, $bud^\#$ and the edge marked #.

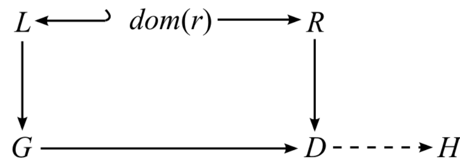


Figure 3.27: SPO direct derivation overview.

Example 3.2.3 (edNCE). *Let* $G = (\{g_0, g_1, g_2, g_3, g_4\}, \{(g_0, >, g_1), (g_0, +, g_3), (g_1, >, g_2), (g_3, >, g_4)\}, G_\lambda)$ *be a graph over the alphabet* $\Lambda = (\{bud, gu\}, \{>, +\})$. G_λ *specifies the labelling of the nodes, where* $G_\lambda(g_0) = G_\lambda(g_1) = G_\lambda(g_3) = gu$, *and* $G_\lambda(g_2) = G_\lambda(g_4) = bud$. *We define an edNCE production* $p : L \overset{e}{\to} R$ *for* G, *where* L *is a graph* $(\{l_1\}, \{\}, L_\lambda : l_1 \mapsto bud)$, *and* R *is*

a graph $(\{r_0, r_1, r_2, r_3, r_4\}, \{(r_0, >, r_1), (r_0, +, r_3\ ), (r_1, >, r_2), (r_3, >, r_4)\}, R_\lambda)$ *where* $R_\lambda(r_0) = R_\lambda(r_1) = R_\lambda(r_3) = gu$, *and* $R_\lambda(r_2) = R_\lambda(r_4) = bud$. *e consists of only one connection instruction,* $(l_0, gu, >\ /\ >, r_0, in)$. *The graphs and production are illustrated in Figure 3.28.*

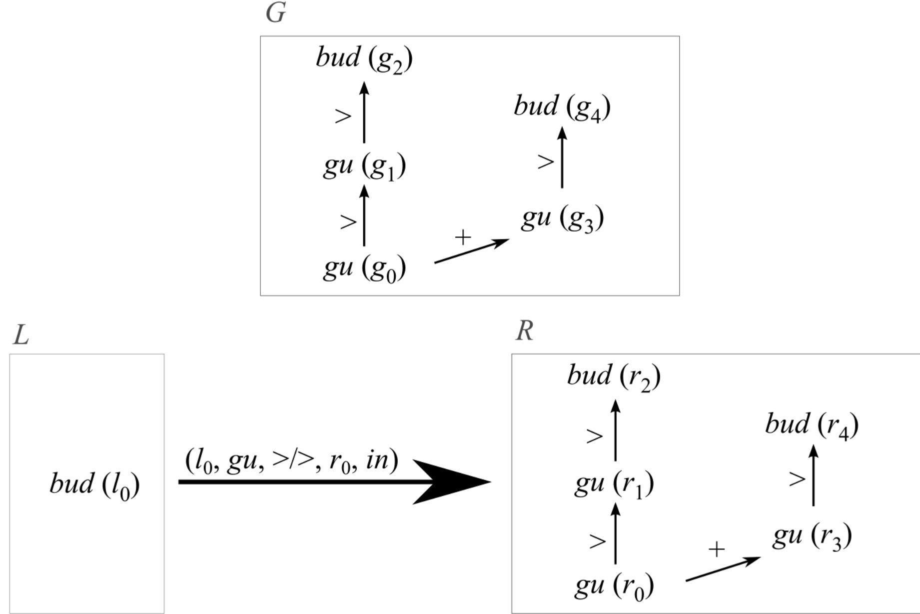


Figure 3.28: edNCE graphs and production. Top: host graph G. Bottom: graphs L and R belonging to production p. The connection instruction $(l_0, gu, >\ /\ >, r_0, in)$ is specified on the arrow from L to R.

Given a match m *where* $m(l_0) = g_2$, *the node* g_2 *and its incident edge is removed, resulting in the remaining host graph* G^-. *A direct derivation of* p *is made by making an isomorphic copy of* R, R^c, *which is joined with* G^- *using the connection instruction. Figure 3.29 illustrates this direct derivation. Notice that another possible match in this example can be made on the node* g_4. *We will discuss parallel rewriting in Section 3.3, where one direct derivation can account for such cases of multiple matches.*

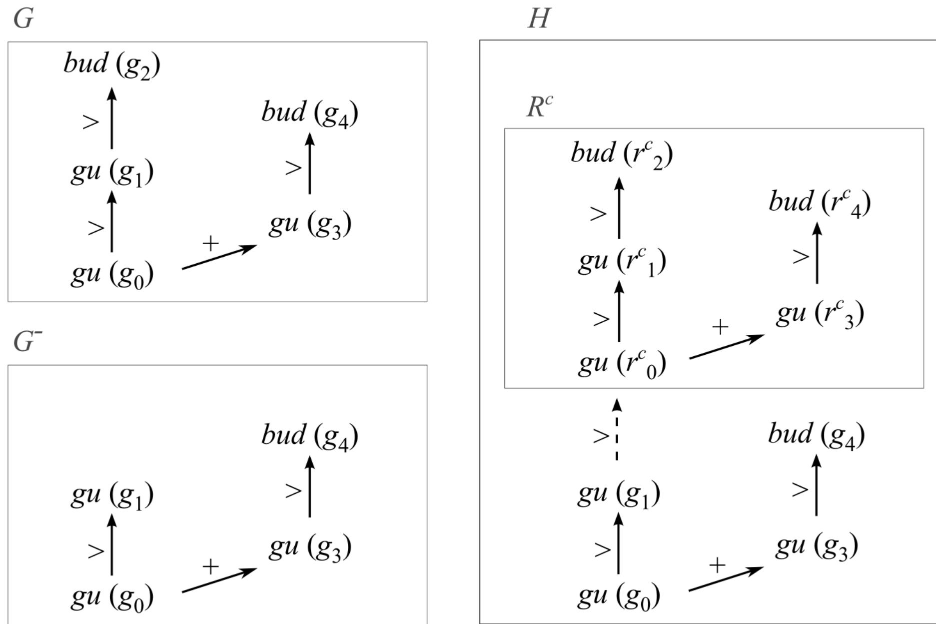


Figure 3.29: Derivation of edNCE production. Top left: host graph G. Bottom left: remaining host graph G^{-} with g_2 and its incident edge removed. Right: R^c is an isomorphic copy of R, where r_i^c are copies of r_i. The dotted edge is established by the connection instruction in e. H is the result of the direct derivation.

3.3 Graph Rewriting in XL

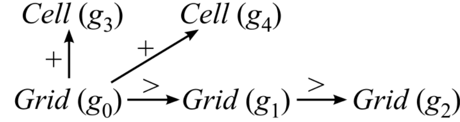
The approaches in Section 3.2 offer glimpses of graph rewriting methods. In this section, we present the two main graph rewriting approaches (chapter 5.3 in [91]) in relational growth grammars (RGG) [92], the underlying formalism of the XL programming language. Unlike the sequential approaches in Section 3.2, both of these techniques are parallel in nature, the first intended for general cases, and the second to provide a specific L-system-like form of rewriting.

3.3.1 Parallel Single-Pushout (SPO) Approach

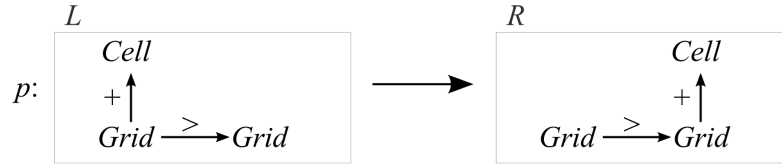
The sequential SPO approach in Section 3.2.2 was considered for XL because its properties apply well in the context of biological models (cf. page 104 in

[91]). For example, SPO rewriting removes dangling edges, a trait suitable for the removal of graph nodes representing dying plant organs. In addition, because node deletion is considered a deliberate action in models intended to be developed in XL (e.g. a production for death takes precedence over a production for metabolic activity), the SPO approach overcomes conflicts arising from the identification condition of the DPO approach (see Section 3.2.1). However, there are cases where parallel productions are necessary, e.g. for the modelling of concurrent cell movements using a single production. For this purpose, the parallel SPO approach [48] is used in XL. We describe this approach here using an example.

Example 3.3.1 (**Parallel SPO**). *Consider a graph* $G = (\{g_0, g_1, g_2, g_3, g_4\}, \{(g_0, >, g_1), (g_0, +, g_3), (g_0, +, g_4), (g_1, >, g_2)\}, G_\lambda)$ *over the alphabet* $\Lambda = (\{Cell, Grid\}, \{>, +\})$. G_λ *labels the nodes such that* $G_\lambda(g_0) = G_\lambda(g_1) = G_\lambda(g_2) = Grid$ *and* $G_\lambda(g_3) = G_\lambda(g_5) = Cell$*, i.e. three of them represent places in a regular spatial grid and two of them represent cells. An illustration of G looks like:*

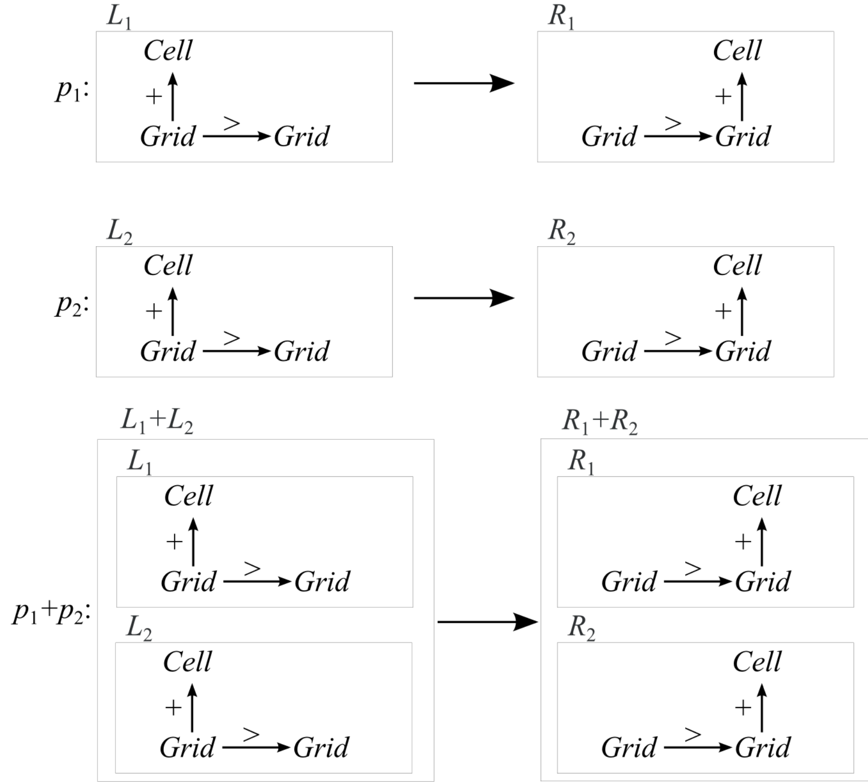


To model the movement of a cell to the neighbouring grid place, a production $p : L \to R$ *is specified:*



Two matches based on p occur, one for each cell in G. We call the match with $Cell(g_3)$ m_1 *and the match with* $Cell(g_4)$ m_2*.*

A parallel derivation of m_1 *and* m_2 *proceeds in two main steps. (This two step approach relates to the general* amalgamated two-level derivation *technique in [50]). In the first step, independent versions of the production for* m_1 *and* m_2*,* $p_1 : L_1 \to R_1$ *and* $p_2 : L_2 \to R_2$ *respectively, are combined using co-product (see Section 3.1.2.10) operations. We illustrate this first combination step below:*



where $L_1 + L_2$ is a co-product of L_1 and L_2, and $R_1 + R_2$ is a co-product of R_1 and R_2. A combined production $p_1 + p_2 : L_1 + L_2 \to R_1 + R_2$ is formed.

In the second step, the derived graph H is obtained using $p_1 + p_2$ by the sequential SPO approach described in Section 3.2.2. We illustrate this derivation in Figure 3.30.

3.3.2 L-system-style Connection

While the parallel SPO approach (Section 3.3.1) supports general cases, RGG includes a type of graph rewriting that mimics the operation of L-system rules (cf. chapter 5.3 in [91]). In this section, two operator-based approaches are first described in Sections 3.3.2.1 and 3.3.2.2 before concluding in Section 3.3.2.3 with the method adopted in RGG.

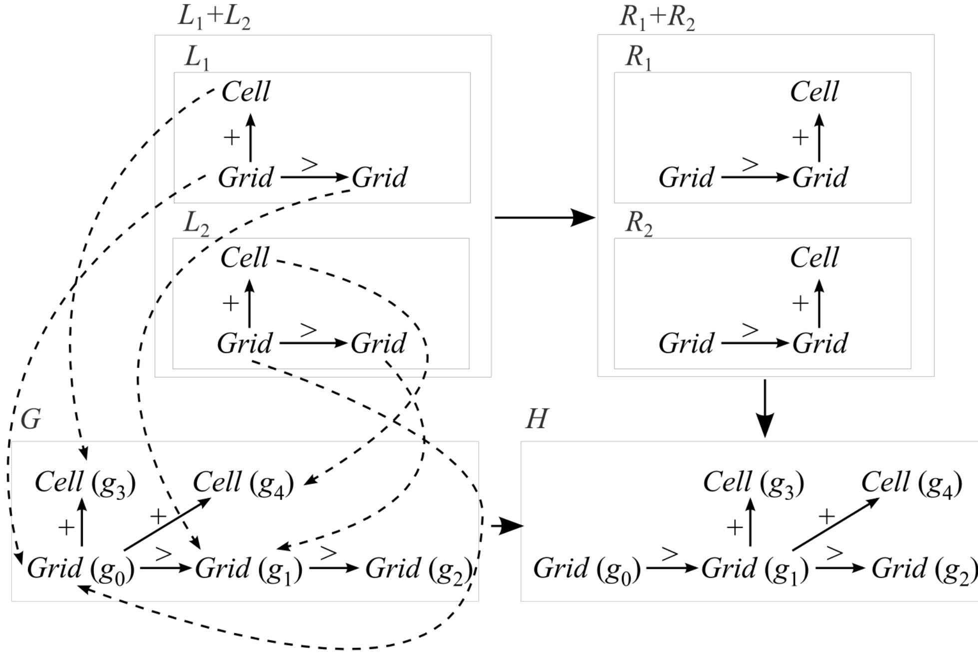


Figure 3.30: Parallel SPO derivation: step 2. Top left: combined left-hand side graph $L_1 + L_2$ for matches m_1 and m_2. Top right: combined right-hand side graph $R_1 + R_2$ for matches m_1 and m_2. Bottom left: host graph G. Bottom right: derived graph H computed as apex of pushout. Dashed arrows curved to the left represent homomorphisms of match m_1 and dashed arrows curved to the right represent homomorphisms of match m_2. $L_1 + L_2$, $R_1 + R_2$, G, H, and the homomorphisms between them form a commutative square.

3.3.2.1 Operator-based Graph Rewriting

Operator-based graph rewriting [124] is an approach that uses *operators*, which identify a set of related nodes for each node in the host graph. We describe here the functionality of this early approach with an example based on [91].

An *operator* is a family of mappings $A_G : G_V \to G_V$ for a graph G such that $n \notin A_G(n)$ for all $n \in G$. A *production with operators* $p : \mu \xrightarrow{\sigma,\tau} R$ is given by a node label μ, a graph R, and two finite sets of connection transformations $c = (A, \gamma, w)$ where A is an operator, $\gamma \in \Lambda_E$ is an edge

label, and $w \in R$ is a node of the right-hand side graph. A direct derivation of host graph G to derived graph H using p exists if a match occurs for each node $v \in G_V$, and H consists of $\sqcup_{v \in G_V} R^v$ (where R^v is an isomorphic copy of R for a match on v) and a set of additional edges E. Let an edge in E be $(s \in H_V, \gamma, t \in H_V)$. s and t are nodes produced based on matched nodes $s_p \in G_V$ and $t_p \in G_V$ respectively. For the direct derivation to hold, there must exist connection transformations $\alpha = (S, \gamma, s) \in \sigma$ and $\beta = (T, \gamma, t) \in \tau$, where $t_p \in S(s_p)$ and $s_p \in T(t_p)$.

Example 3.3.2 (Parallel graph rewriting with operators). *Consider a graph* $G = (\{g_0, g_1\}, \{(g_0, >, g_1)\}, G_\lambda : g \mapsto gu)$ *over an alphabet* $\Lambda = (\{gu\}, \{>\})$. *We define a production with operators* $p : gu \xrightarrow{\sigma,\tau} R$, *where* $R = (\{r_0, r_1\}, \{(r_0, >, r_1)\}, R_\lambda : g \mapsto gu)$. σ *consists of a connection instruction* $(A^{out}, >, r_1)$, *and* τ *consists of a connection instruction* $(A^{in}, >, r_0)$. A^{out} *is an operator that returns all nodes* $v_{out} \in G_V$ *with an edge connection* $(n, >, v_{out})$ *from a matched node* n. A^{in} *is an operator that returns all nodes* $v_{in} \in G_V$ *with an edge connection* $(v_{in}, >, n)$ *to a matched node* n.

A direct derivation proceeds as follows: g_0 *and* g_1 *are both matched by the production* p *(resulting in matches* m_0 *and* m_1 *respectively). They are removed from* G, *resulting in an empty remaining host graph* G^-. *Two isomorphics copies of* R *are produced,* $R^{c0} = (\{r_0^{c0}, r_1^{c0}\}, \{(r_0^{c0}, >, r_1^{c0})\}, R_\lambda^{c0})$ *and* $R^{c1} = (\{r_0^{c1}, r_1^{c1}\}, \{(r_0^{c1}, >, r_1^{c1})\}, R_\lambda^{c1})$ *corresponding to* m_0 *and* m_1 *respectively. The operator* A^{out} *yields* $\{g_1\}$ *for* m_0 *and* $\emptyset$ *for* $m1$, *while the operator* A^{in} *yields* $\{g_0\}$ *for* m_1 *and* $\emptyset$ *for* $m0$. *Consequently, an edge labelled* $>$ *is established between* r_1^{c0} *and* r_0^{c1}. *Figure 3.31 illustrates this direct derivation process.*

3.3.2.2 Operator-based L-system-style Graph Rewriting

To utilize the operator-based approach in Section 3.3.2.1 for L-system-style graph rewriting, Kniemeyer first extended the notion of well-nested words [142] to translate strings to graphs (cf. page 95 in [91]). The translation is based on axial tree representations ([144], [67]) of strings in L-system rules. It is essential for the identification of nodes in the right-hand side graphs of productions, so that they establish embedding edges in a manner resembling L-system derivations.

Based on [91], a well-nested word is a word that can be generated by the context-free grammar

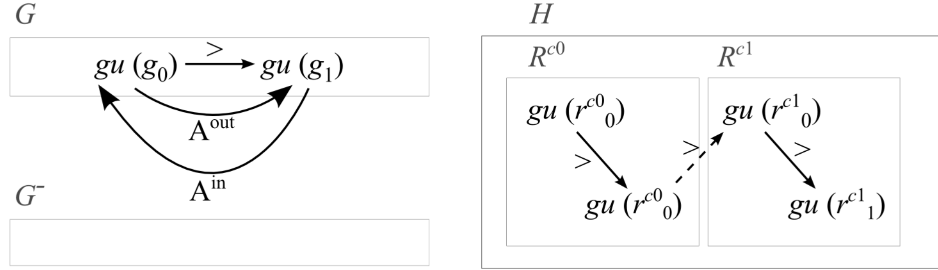


Figure 3.31: Derivation of production with operators. Top left: host graph G. Arrows with large heads represent a complementing pair of operator mappings, $A^{out}(g_0) = g_1$ and $A^{in}(g_1) = g_0$. Bottom left: empty remaining host graph G^-. Right: Derived graph H consisting of isomorphic copies of R, R^{c0} and R^{c1}, and the dotted edge established based on the complementing pair of operator mappings seen at the top left.

$$\begin{gathered}
\alpha \rightarrow A_0, \\
A_0 \rightarrow \lambda, \\
A_0 \rightarrow A, \\
A \rightarrow aA_0 \ \forall a \in \Lambda, \\
A \rightarrow \% A_0, \\
A \rightarrow [A_0]A_0.
\end{gathered}$$

In this grammar, α is the axiom, A is a non-empty well-nested word, A_0 is a possibly empty well-nested word, λ is the empty word, % is the *cut-operator* turtle command [4], and Λ is an alphabet of symbols excluding [,], and %. It allowed the specification of a recursive function that translates strings in the form of well-nested words to graphs (see specification of function on page 96 in [91]) by parsing from left to right. We illustrate the translation using an example:

Example 3.3.3 (Translation of well-nested words). *Consider a well-nested word $[A[[B\%CD]E[F]]G][H]I$. Upon translation, the word yields an unconnected graph G, a set of left-most nodes L, a set of right-most nodes R, and a set of pending branch nodes B. G is given by $(\{a, b, ..., i\}, \{(a, +, b), (c, >, d), (a, +, e), (e, +, f), (a, >, g)\}, G_\lambda)$, where $G_\lambda(a) = A$, $G_\lambda(b) = B$, ..., $G_\lambda(i) = I$. Both L and R consist of one node i, considering that I is the left-*

most and right-most symbol not enclosed in brackets. Lastly, B contains the nodes a and h, which are the left-most nodes within brackets. We illustrate the resulting graph in Figure 3.32.

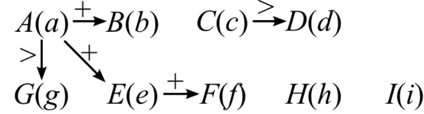


Figure 3.32: Graph translation of a well-nested word (a string in an L-system rule) $[A[[B\%CD]E[F]]G][H]I$. $I(i)$ is in the sets of left-most and right-most nodes, L and R respectively. The set of pending branch nodes B contains the nodes $A(a)$ and $H(h)$.

Based on this translation of well-nested words, a D0L-system rule can be translated to a special case of graph rewriting production with operators (cf. [91]). Consider a D0L-system rule $r : a \to \chi$ over an alphabet V, where $a \in V$ and χ is a well-nested word. We specify the translation function of well-nested words to graphs as $T : \chi \mapsto (G, L, R, B)$, where G is the graph translation of χ, and L, R, and B are the sets of left-most, right-most, and pending branch nodes respectively. r is translated to a graph rewriting production with operators $p : a \xrightarrow{\sigma,\tau} G$ with

$$\sigma = \bigcup_{\gamma \in \Lambda_E, s \in R \setminus \{\%\}} \{(N_\gamma^{out}, \gamma, s), (N_\gamma^{out}, +, s)\}$$
$$\tau = \bigcup_{\gamma \in \Lambda_E, t \in L \setminus \{\%\}} \{(N_\gamma^{in}, \gamma, t)\} \cup \bigcup_{\gamma \in \Lambda_E, t \in B} \{(N_\gamma^{in}, +, t)\} \ .$$

In this production, $\Lambda = (\Lambda_V, \Lambda_E)$ is the alphabet of node and edge labels. The operator N_e^{dir} returns the set of nodes connected to a matched node via an edge labelled e in the direction dir. The following example illustrates the operation of such a production.

Example 3.3.4 (L-system-style production with operators). *Consider a string, $a\ a\ a$. To differentiate between the three symbols, we label them with indices so the string becomes $a_1\ a_2\ a_3$. A D0L-system rule $r : a \to [b]c$ is applied to the string to derive $[b_1]c_1[b_2]c_2[b_3]c_3$, the indices indicating the original symbol before rule application. We describe this rule derivation in graph grammars using the above special case of production with operators:*

The string, a a a, is represented as a graph $G = (\{a_1, a_2, a_3\}, \{(a_1, >, a_2), (a_2, >, a_3)\}, G_\lambda : a_i \mapsto a)$ *over the alphabet* $\Lambda = (\{a, b, c\}, \{>, +\})$. *In order to specify connection instructions, the well-nested word,* $[b]c$, *on the right-hand side of rule* r *is translated to* $(K, L, R, B) = (K, \{c\}, \{c\}, \{b\})$, *where* $K = (\{k_1, k_2\}, \emptyset, K_\lambda)$ *is a graph over* Λ, $K_\lambda(k_1) = b$, *and* $K_\lambda(k_2) = c$. *To transform* G *like the rewriting of the string by* r *above, a production with operators* $p : a \xrightarrow{\sigma,\tau} K$ *is specified.*

A match occurs for each of the three nodes in G, *and isomorphic copies of* K, K^{c1}, K^{c2}, K^{c3}, *are made for matches on* a_1, a_2, *and* a_3 *respectively. Figure 3.33 illustrates* G, *the production* p, *and the copies of* K *(i.e.* K^{c1}, K^{c2}, K^{c3}*) made for each match.*

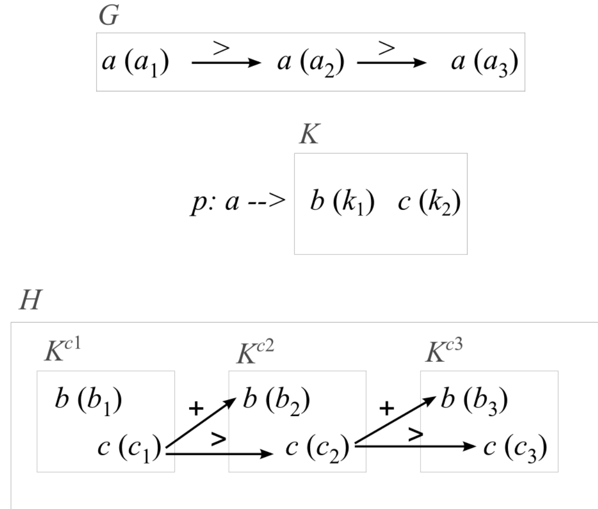


Figure 3.33: L-system-style production. Top: original graph G before derivation. Middle: production p with graph K on right-hand side. Bottom: Derived graph H with isomorphic copies of K for each match and embedding edges.

We proceed to describe the sets of connection instructions, σ *and* τ *for the derivation.* σ *consists of the connection instructions:*

$$s_1 = (N_>^{out}, >, c),$$
$$s_2 = (N_>^{out}, +, c), \text{ and}$$
$$s_3 = (N_+^{out}, +, c).$$

τ consists of the complementing connection instructions:

$$s_4 = (N^{in}_{>}, >, c),$$
$$s_5 = (N^{in}_{+}, +, c),$$
$$s_6 = (N^{in}_{>}, +, b),$$
$$s_5 = (N^{in}_{+}, +, b).$$

Using these connection instructions and their operators, half-edges pending connections are created. For the match on a_1, two half-edges:

$$e_1 = (c_1, >, ?) \text{ and}$$
$$e_2 = (c_1, +, ?)$$

pending connections to K^{c2} are created based on s_1 and s_2 respectively. For the match on a_2, four half-edges:

$$e_3 = (c_2, >, ?),$$
$$e_4 = (c_2, +, ?),$$
$$e_5 = (?, >, c_2),$$
$$e_6 = (?, +, b_2),$$

pending connections to K^{c3} and from K^{c1} are created. Lastly, for the match on a_3, two half-edges:

$$e_7 = (?, >, c_3),$$
$$e_8 = (?, +, b_3),$$

pending connections from K^{c2} are created. The pairs of complementing half-edges result in the embedding edges in H as shown in Figure 3.33:

- *e_1 and e_5: $(c_1, >, c_2)$*
- *e_2 and e_6: $(c_1, +, b_2)$*
- *e_3 and e_7: $(c_2, >, c_3)$*
- *e_4 and e_8: $(c_2, +, b_3)$*

Although this special form of production with operators was defined to mimic rules in D0L-systems, it is mentioned in [91] that extentions to other forms of L-systems are straightforward and require only more complex notations.

3.3.2.3 Single-pushout (SPO) with Operators

This section describes the second rewriting approach called *single-pushout (SPO) with operators* (cf. page 104 in [91]) in RGG and XL for L-system-style connections. It combines the functionalities of the operator-based L-system-style rewriting in Section 3.3.2.2 and the parallel SPO approach in Section 3.3.1. We illustrate the operation of this approach with an example, which will reveal the reasons for such a combination.

Example 3.3.5 (SPO with operators). *Consider a graph* $G = (\{g_0, g_1, g_2\}, \{(g_0, >, g_1), (g_1, >, g_2)\}, G_\lambda)$ *over an alphabet* $\Lambda = (\{A, B, C\}, \{>, +\})$. G_λ *labels the nodes such that* $G_\lambda(g_0) = G_\lambda(g_1) = A$ *and* $G_\lambda(g_2) = C$. *A diagram of* G *appears as follows:*

$$A\,(g_0) \xrightarrow{>} A\,(g_1) \xrightarrow{>} C\,(g_2)$$

An SPO production with operators $p : L \xrightarrow{\sigma,\tau} R$ *is specified to rewrite* G *in order to obtain a derived graph* H. L *and* R *are represented as translated well-nested words in the forms* $L = (L_G, L_L, L_R, L_B)$ *and* $R = (R_G, R_L, R_R, R_B)$ *respectively. The subscripts* $_G$, $_L$, $_R$, $_B$ *represent the graph, the left-most, the right-most, and the pending branch nodes respectively (see Example 3.3.3 for the translation of well-nested words).* L_G *is a graph* $(\{l_0\}, \emptyset, L_{G\lambda}(l_0) = A)$ *and* R_G *is a graph* $(\{r_0, r_1, r_2, r_3\}, \{(r_0, >, r_1), (r_1, >, r_2), (r_1, +, r_3)\}, R_{G\lambda})$, *where* $R_{G\lambda}(r_0) = A$ *and* $R_{G\lambda}(r_1) = R_{G\lambda}(r_2) = R_{G\lambda}(r_3) = B$. *A diagram of* p *appears as follows:*

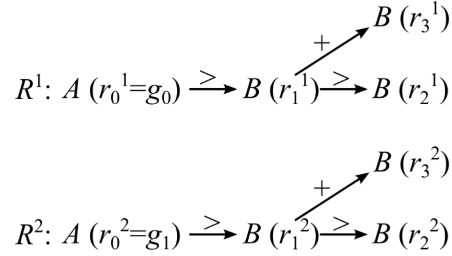
$$p\colon\ A\,(l_0) \longrightarrow A\,(r_0) \xrightarrow{>} B\,(r_1) \xrightarrow{>} B\,(r_2), \qquad B\,(r_1) \xrightarrow{+} B\,(r_3)$$

The dotted arrow represents a homomorphism from node l_0 *to* r_0. *Based on this setup,* $L_L = \{l_0\}$, $L_R = \{l_0\}$, $L_B = \emptyset$, $R_L = \{r_0\}$, $R_R = \{r_2\}$, *and* $R_B = \emptyset$. σ *and* τ *are sets of connection instructions as previously described*

in Section 3.3.2.2. However, the operator N_γ^{out} returns nodes connected to the matched node of the node in L_R via out-going edges labelled γ, and the operator N_γ^{in} returns nodes connected to the matched node of the node in L_L via incoming edges labelled γ. The derivation of graph H proceeds in five steps:

Step 1*:*

For each match m^i ($1 \leq i \leq n$, n being the number of matches), an isomorphic copy R^i of R is derived using the sequential SPO method described in Section 3.2.2. In this example, there is a match on g_0 and another on g_1, resulting in the following R^i graphs:



Step 2*:*

The embedding edges are determined following the operator-based approach in Section 3.3.2.2. For the match m^1, the half-edge

$$e_0 = (r_2^1, >, ?)$$

pending connection to R^2 is created. For the match m^2, the half-edges

$$e_1 = (r_2^2, >, (g_2)?) \text{ and}$$
$$e_2 = (?, >, r_0^2)$$

pending connections to g_2 and R^1 respectively are created.

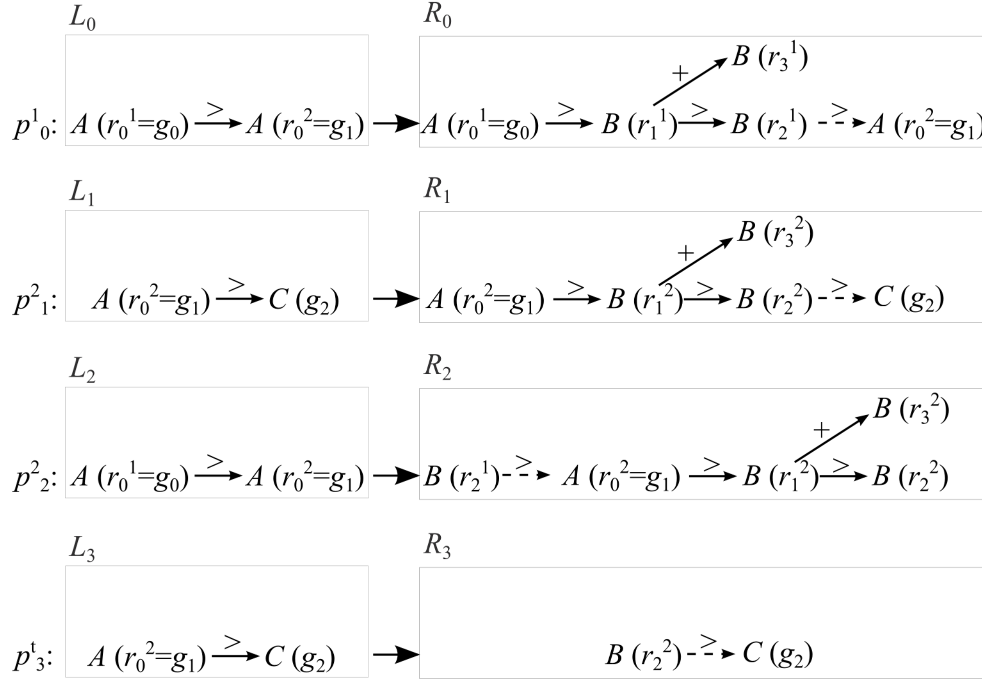
Remark: Although the operator resulting in e_1 returned g_2, which is a node in G that is not matched, the half-edge is still created as a trivial (non-parallel, sequential) case for embedding in the remaining host graph G^-. However,

such half-edges are not created for connection instructions in σ that are in the form $(N_\gamma^{out}, +, s)$. This allows sequential embedding similar to the edNCE *approach.*

A pair of complementing half-edges (e_0, e_2) and the half-edge e_1 with no complement synthesizes the embedding edges $(r_2^1, >, r_0^2 = g_1)$ and $(r_2^2, >, g_2)$ respectively.

Step 3*:*

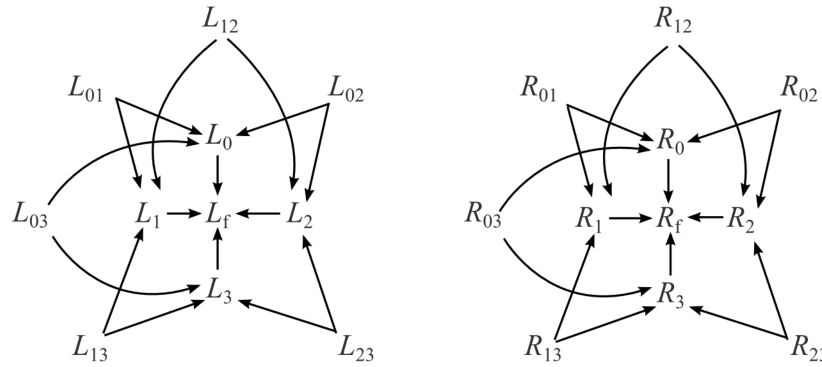
For each embedding edge and each match, a new SPO production $p_j^i : L_j \to R_j$ is specified to include the gluing node (either the source or target of the embedding edge), where i corresponds to match m^i and j corresponds to embedding edge e_j. The following productions are created:



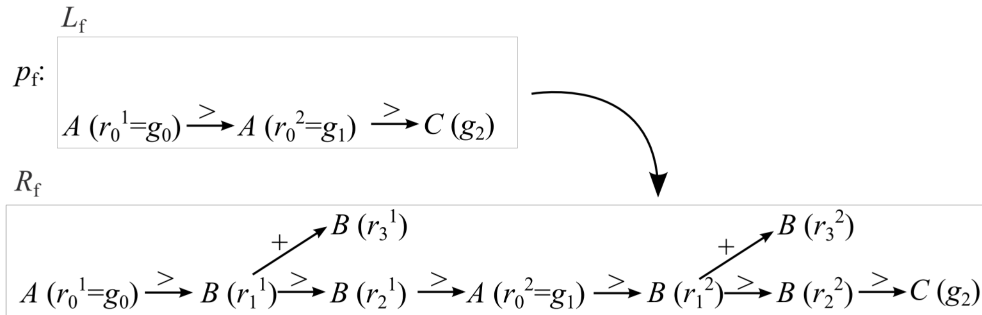
The dashed edges in R_j represent the embeddings edges created in the previous step. This step is a preparation step for an application of the amalgamated two-level derivations (see page 65 in [91]) for parallel SPO in the next two steps.

Step 4:

The four productions in Step 3 are amalgamated using pushout-stars (cf. page 63 [91]) to obtain a final production $p_f : L_f \to R_f$. This is the first level of an amalgamated two-level derivation. We illustrate the commutative diagrams (pushout-stars) for obtaining L_f and R_f below:



Each L_{ij} consists of the common nodes and edges in L_i and L_j, and each R_{ij} consists of the common nodes and edges in R_i and R_j. As a result, the final production p_f is:



Step 5:

Finally, the production $p_f : L_f \to R_f$ is used for a SPO direct derivation of H from G. We illustrate this derivation in Figure 3.34.

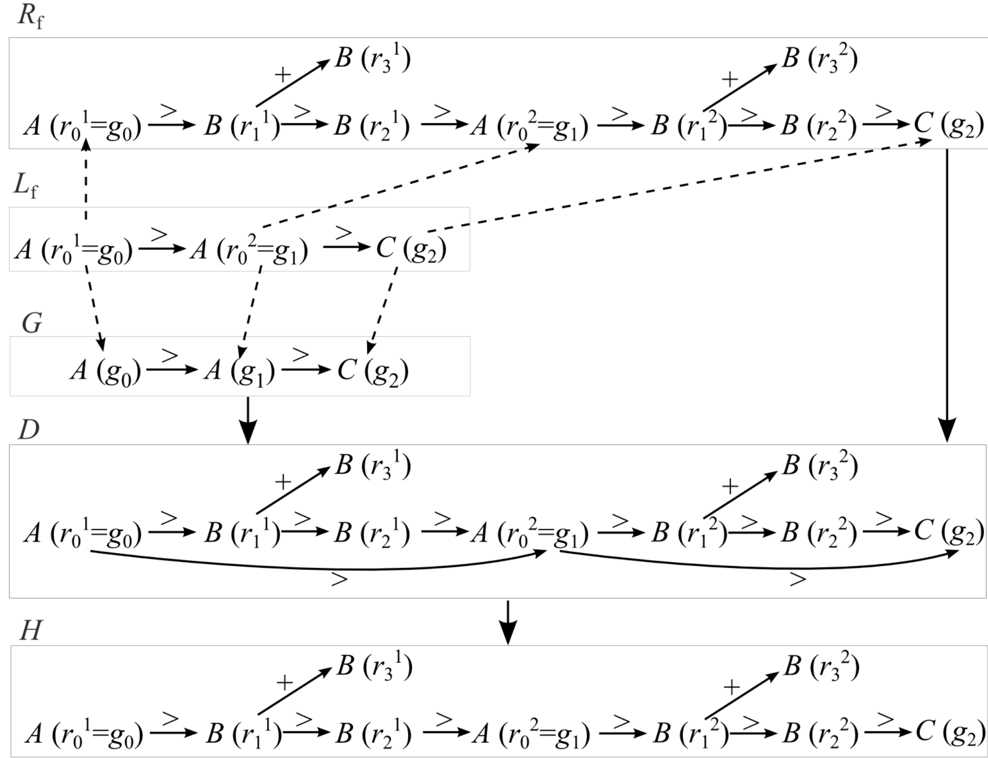


Figure 3.34: SPO derivation with operators. L_f and R_f are the left-hand side and right-hand side graphs of the amalgamated production p_f obtained in the previous step of the SPO with operators method. The dashed arrows from L_f to R_f represent a partial homomorphism. The dashed arrows from L_f to G represent a match that is a total homomorphism. $(dom(p_f) \subseteq L_f) \to R_f$ is a total homomorphism corresponding to $L_f \to R_f$. D is computed as the apex of the pushout by arrows $dom(p_f) \to R_f$ and $dom(p_f) \to G$. The derived graph H as a result of applying p_f to G is obtained by a coequalizer operation, in which the edges $(g_0, >, g_1)$ and $(g_0, >, g_2)$ are deleted. Arrows with large heads are collective representation of graph homomorphisms to avoid cluttering.

Step 1 in Example 3.3.5 shows the necessity of gluing which is not catered for by the operator-based approach in Section 3.3.2.2. Step 2 in the same example highlights the dynamic identification of desired neighbouring nodes for each match using operators, a feature absent in the parallel SPO approach

in Section 3.3.1. The combined approach delivers the functionality of both approaches and allows an L-system-style connection in RGG.

In this chapter, we have described the graph rewriting formalisms in RGG, which underlies the XL programming language. In the next chapter, we introduce new techniques and extensions of XL as part of this thesis for multiscale modelling.

Chapter 4

XL for Multiscale Modelling

This chapter introduces new concepts that address the first part of the thesis topic - "Extension of the Rule-Based Programming Language XL by Concepts for Multi-Scaled Modelling". The sections contain information published in [128] and [130].

To express the modularity and multi-scale nature of plant topology, Godin et al. introduced the Multiscale Tree Graph (MTG) [68]. MTGs allow the preservation of plant structural information at several scales, e.g. internodes, axes or branching systems (see Figure 4.1). In addition to spatial scales, variations in MTGs recorded at particular time instances reflect incremental growth of plant topology [67]. In 2012, an integration of L-systems with the Python language called L-Py [14] was made with the capability to represent MTG data as multiscale L-system strings. String-based rules can be applied on these strings using scripts in L-Py.

As we have seen in the previous chapters, parallel graph grammars (utilized in relational growth grammars (RGG)) are considered a generalization of string-based L-systems [38, 62]. Therefore, we are motivated to develop a data structure and rewriting method that retain the versatility of graphs while allowing the specification of a dynamic model with multiple scales. To keep the implementation flexible for a wider variety of modelling requirements, we do not restrict our research on tree structures, i.e. MTGs, so that other graph structures can be represented. It is our intention that the proposed model and grammar are suitable with single-scale models as well as existing ones built on RGG [20, 92, 93]. *They should be well adapted but not limited to applications in FSPM.* In Section 4.1, a generalized framework to conduct multi-scale modelling using XL is given. Concrete developments

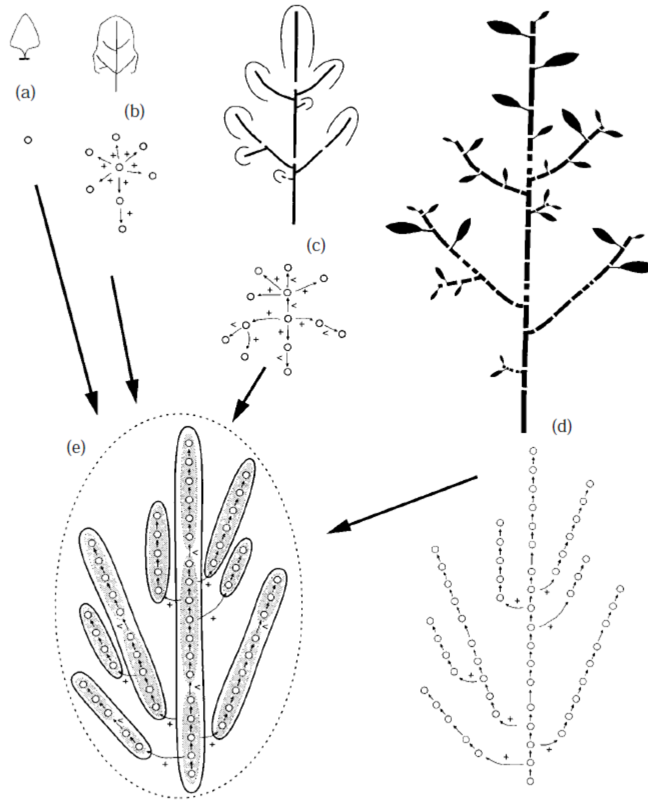


Figure 4.1: Multiscale tree graph (MTG): a tree at different scales of perception (leaves not taken into account): (a) Tree scale, (b) axis scale, (c) growth unit scale, (d) internode scale, (e) corresponding multiscale tree graph. Figure extracted from [67].

are documented in Sections 4.2, 4.3, and 4.4. Section 4.5 gives a description of syntax changes in the XL programming language that support and provide access to the new data structure and rewriting formalism. To conclude the chapter, technical details of implementation in the software GroIMP are documented in Section 4.6.

4.1 Multiscale Modelling Framework

We begin first by addressing the dynamic aspects of modelling from a broad perspective. Considering the hierarchy of physical models (ranging from

continuum models to quantum mechanics), the interest of a modeller usually resides in a predominant scale. If the scale of interest is macroscopic, the effects of inputs to the microscopic model are usually modelled by some constitutive correlations at the macro scale because representative models at the micro scale often pose computational or analytical problems [46]. These correlations are usually obtained empirically. However, a correlation-based model is soon loaded with parameters with obscure meaning as it becomes increasingly complex with more microscopic inputs. For example, consider constructing a model for the structural growth of trees under the effects of an atmospheric component. One possible way could be an experimental observation of structural growth in relation to gas concentrations, followed by a construction of a mathematical model after data interpretation. It would be computationally impractical to model particular responsive metabolic networks in each cell of the trees. Despite their success in many applications, the extension of such correlation-based approaches to complex scenarios has proven to be difficult, often requiring complicated mathematical functions. E [46] illustrated this argument with extensions of the Navier-Stokes equation (which commonly uses an empirically obtained stress tensor parameter) for complex fluids such as polymeric fluids [11]. The quantum mechanics-molecular mechanics (QM-MM) model of chemical reactions [175] and the first-principle-based molecular dynamics [22] are two examples of successful multiscale applications that overcame these difficulties. Subsequently, several general frameworks for multiscale modelling in mathematical physics such as the heterogeneous multiscale method (HMM) by [47] have been developed. These frameworks from domains of science relatively distant from FSPM, nevertheless, offer a concise overview of multiscale concepts. They address a fundamental difficulty in mathematical modelling inherent to correlation-based approaches.

The three-part graph data structure which will be introduced in Section 4.3 offers only an infrastructure for modelling [130]. To justify its utility, we propose a multiscale modelling framework inspired by the extended multi-grid method [16], the equation-free approach [89] and the heterogeneous multiscale method (HMM; [47], [46]). This approach consists of three components, namely, *problem categorization*, *scale-dependency*, and *scale integration*.

The *problem categorization* component adopts the two categories of multiscale problems introduced by E [46]. Both problem categories are characterized by significant disparities of simulation output from expected output.

One category (hereafter referred to as "local") is characterized by disparities at localized regions in the domain of the coarse scale. The fine scale is employed to resolve the disparities occurring at these regions. As a hypothetical example, an FSPM of tree growth at organ scale may be integrated with a micro-scale biophysical model of xylem vessels to predict sudden embolisms that can cause a catastrophic dysfunction of the water supply system in a branch or in a whole crown part (cf. [34]). The second category (hereafter referred to as "global") of problems requires the fine scale to overcome disparities throughout the domain of the coarse scale. For example, the production of new internodes and buds may be an extrapolation of meristem cell differentiation throughout the vegetation period. This form of categorization is helpful for identifying and describing the aim of a multiscale model. Contrasting with the problem types by E [46], our framework does not mandate the occurrence of disparities only in the coarse scale. I.e., disparities in the fine scale domain can also be resolved with feedback from coarse scale models.

The notion of *scale dependency* is closely related to problem categorization. The scale with disparities, either locally or throughout its entire domain, is said to be scale dependent on the scale providing information that alleviates the disparities. The scale that provides information to alleviate disparities is said to be scale independent. *Scale integration* serves as a template for scale-to-scale interactions during simulation. It comprises three steps between two comparable scales that are executed in the order of *initialization*, *evolution* and *extrapolation* (Figure 4.2 shows a schematic representation of the steps). Suppose X is a scale dependent on scale Y. If Y is a finer scale than X, it is usually simulated at spatial and time scales much smaller in magnitude than the model in X. While correlation-based approaches seek to avoid simulation at scale Y due to computational or analytical impracticality, our approach attempts to perform a feasible simulation at scale Y and estimates the state of X using the results. In initialization, the state of X may first undergo preparatory modifications. Subsequently, the state of Y is initialized with information from the observable state of the model, including the state of X. Evolution refers to the simulation of a model in Y following the initialized parameters. Finally, the state of X is modified based on the evolved state of Y in extrapolation. We revisit this framework in Chapters 6.4 and 6.5 where its use is demonstrated.

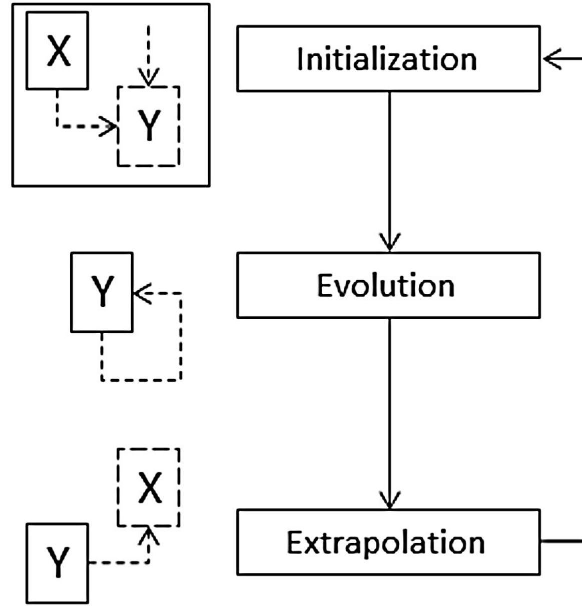


Figure 4.2: Multiscale scale integration steps *initialization*, *evolution*, *extrapolation*. X is a dependent scale and Y is an independent scale. Dotted arrows are directed at scales with state modification (excluding preparatory modifications of X in initialization).

4.2 Statement of Problem

In this section, we introduce the fundamental problem in graph rewriting within the context of RGG when scales are introduced.

Consider a graph rewriting production $p : L \rightarrow R$ that uses the L-system-style connection mechanism described in Section 3.3.2.3. (Although technically the name for the mechanism is *single-pushout with operators*, we refer to it as *L-system-style connection* to differentiate it from the *parallel SPO* method). To simplify the description, we do not explicitly specify L and R as results of translated well-nested words here (for details on translation of well-nested words, see Section 3.3.2.2). Let L be a graph $(\{l_0\}, \emptyset, L_\lambda(l_0) = A)$ and R be a graph $(\{r_0, r_1, r_2\}, \{(r_1, >, r_2)\}, R_\lambda)$ over the alphabet $\Lambda = (\{A, B, C, D, U, X\}, \{>, +\})$. R_λ labels the nodes in R as $R_\lambda(r_0) = B$, $R_\lambda(r_1) = C$, and $R_\lambda(r_2) = D$. The string representation of production p is:

$$A \text{ ==> } [B]\ C\ D,$$

where ==> is syntax in XL for a production using the L-system-style connection. The graph representation of p is:

L R
p: A (l0) → B (r0) C (r1) —>→ D (r2)

We apply p on a host graph $G = (\{g_0, g_1, g_2\}, \{(g_0, >, g_1), (g_1, >, g_2)\}, G_\lambda)$ over Λ, where $G_\lambda(g_0) = U$, $G_\lambda(g_1) = A$, and $G_\lambda(g_2) = X$. The derived graph H based on the connection mechanism is $(\{h_0, h_1, h_2, h_3, h_4\}$, $\{(h_0, >, h_1), (h_1, >, h_2), (h_2, >, h_3), (h_0, +, h_4)\}, H_\lambda)$, where $H_\lambda(h_0) = U$, $H_\lambda(h_1) = C$, $H_\lambda(h_2) = D$, $H_\lambda(h_3) = X$, and $H_\lambda(h_4) = B$. Graphs G and H are graphically represented as below:

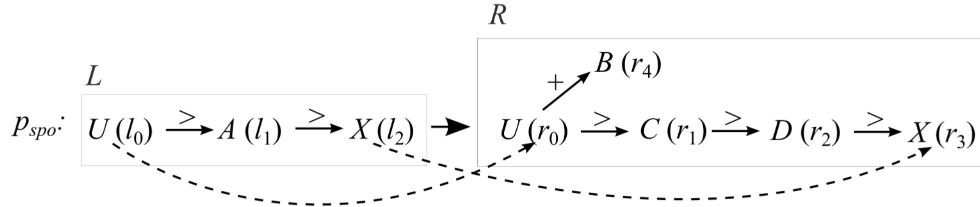
G: U (g0) —>→ A (g1) —>→ X (g2)
p →
H: U (h0) —>→ C (h1) —>→ D (h2) —>→ X (h3); U (h0) —+→ B (h4)

The same rewriting operation can be specified by a production that uses the parallel SPO mechanism described in Section 3.3.1. Such a production is specified as $p_{spo} : L \to R$, where $L = (\{l_0, l_1, l_2\}, \{(l_0, >, l_1), (l_1, >, l_2)\}, L_\lambda)$, and $R = (\{r_0, r_1, r_2, r_3, r_4\}, \{(r_0, >, r_1), (r_1, >, r_2), (r_2, >, r_3), (r_0, +, r_4)\}, R_\lambda)$. L_λ labels the nodes in L as $L_\lambda(l_0) = U$, $L\lambda(l_1) = A$, $L_\lambda(L_2) = X$, and R_λ labels the nodes in R as $R_\lambda(r_0) = U$, $R_\lambda(r_1) = C$, $R_\lambda(r_2) = D$, $R_\lambda(r_3) = X$, $R_\lambda(r_4) = B$. The string representation of production p_{spo} is:

$$u : U \quad A \quad x : X \text{ ==>> } u\ [B]\ C\ D\ x,$$

where ==>> is syntax in XL for a production using the parallel SPO connection mechanism. The small letters u and x are labels attached to matched nodes of U and X respectively. Their prescence in the right-hand side indicate that they are the gluing nodes in the SPO mechanism, i.e. a partial homomorphism maps u and x on the left-hand side to u and x on the right-

hand side respectively. The graph representation of p_{spo} is:



,

where the dotted arrows show the partial homomorphism. The derived graph H obtained by applying p_{spo} to G is the same as that obtained by applying p.

Consider the use of an edge type (symbolized as /) to represent refinement relationships between nodes representing the same entity in the graph model at different scales. The alphabet Λ hence becomes $(\{A, B, C, D, S, U, X\}, \{>, +, /\})$, where S is a node label for a coarser scale representation. We now apply the production p on another version of graph G containing a coarser scale. The new graph G is specified as $(\{g_0, g_1, g_2, g_3\}, \{(g_0, >, g_1), (g_1, >, g_2), (g_3, /, g_0), (g_3, /, g_1), (g_3, /, g_2)\}, G_\lambda)$ over Λ. Figure 4.3 shows graph G and derived graph H via the application of p on G. The L-system style

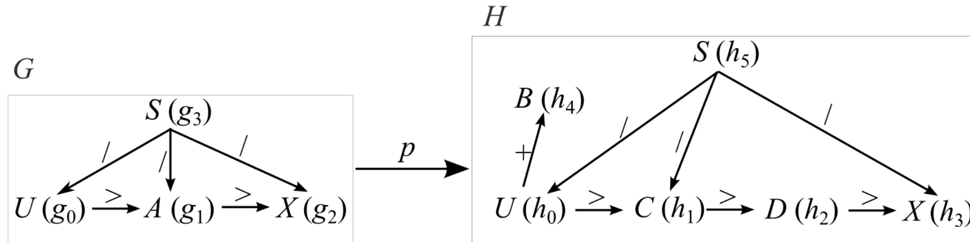


Figure 4.3: L-system style embedding with multiple scales. Edges labelled '/' are refinement (inter-scale) edges. Node S is a coarse scale representation of an entity (e.g. growth unit) comprising of 3 fine scale representations (nodes U, A and X) of the same entity (e.g. internodes). Notice that nodes B and D are not connected to node S after rule execution based on the L-system-style embedding mechanism.

embedding in RGG fails to take the refinement relationships correctly into account. Assuming that B and D are in the same scale as A, they are not

connected by refinement edges from S after derivation although we intuitively expect them to be.

One can of course argue that such connections can be established if we utilize the *parallel SPO* approach and indicate all expected embedding edges explicitly. In order to do so, we modify p_{spo} to include the coarse scale node labelled S:

$$\begin{gathered} u:U \quad a:A \quad x:X, \quad s:S/>u, \quad s/>a, \quad s/>x \\ ==>> \\ u \quad [b:B] \quad c:C \quad d:D \quad x, \\ s\ /> u, \quad s\ /> b, \quad s\ /> c, \quad s\ /> d, \quad s\ /> x \end{gathered}$$

By applying this modified version of p_{spo}, we can obtain a derived graph H with all expected embedding edges. Figure 4.4 shows graph G and derived graph H via the application of the modified p_{spo} on G. However, this

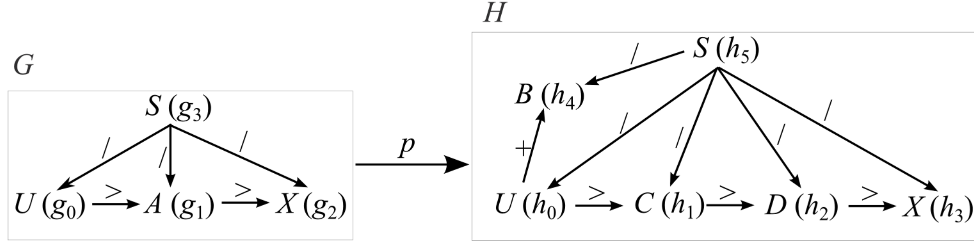


Figure 4.4: SPO embedding with multiple scales. Edges labelled '/' are refinement (inter-scale) edges. Node S is a coarse scale representation of an entity (e.g. growth unit) comprising of 3 fine scale representations (nodes U, A and X) of the same entity (e.g. internodes). '/>' represents a refinement edge. In comparison with Figure 4.3, notice that all inter-scale embedding edges are established.

reveals another problem. The explicit connection specifications at the end of the right-hand side production statement between the coarse scale node S and the finer scale nodes are very lengthy. Considering that only one additional scale is introduced in this case, the production statement will potentially become very long with more than two scales, making multi-scale rule programming tedious in XL.

4.3 Multiscale Graph Data Structure

4.3.1 Structure-of-Scales

A solution of the problems revealed in Section 4.2 first requires a concrete definition of a data structure that contains the refinement edge type. On one hand, the data structure and its operations should take into account compliance with existing multiscale data, such as MTG encoded plant structures. On the other hand, the restrictions imposed by them on the modelling approach should be minimal. The primary restriction a data structure imposes on the modelling approach is the type of refinement ordering it supports. We describe two types of refinement orderings.

Conventionally, the notion of plants as modular organisms [74] has been used as a reference to relate and construct orderings for the modularities of plants [67], i.e. for the types of repeatedly-occurring morphological entities. Orderings represent decomposition relationships between modularities. In the context of this thesis, the terms refinement and decomposition are used as synonyms. A set of pairwise inter-comparable modularities yields a linear, i.e. strict ordering (Figure 4.5A). When a set of modularities has no decom-

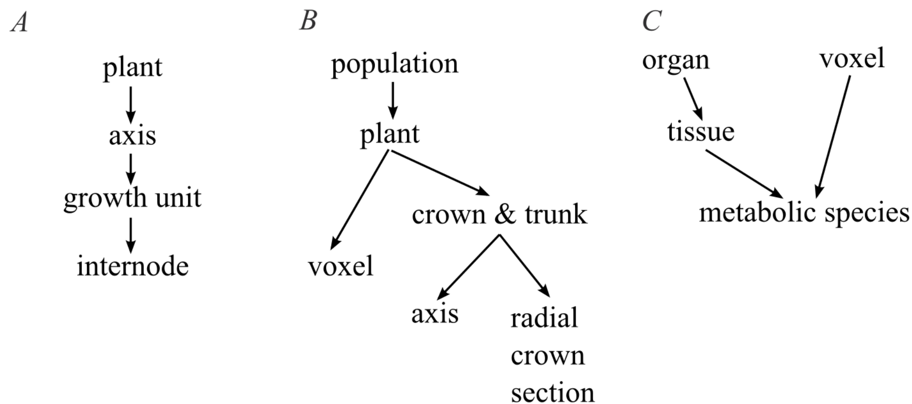


Figure 4.5: Refinement orderings: (A) linear ordering with inter-comparable modularities, (B) fine modularities with common encoarsements and (C) coarse modularities with common refinement. Arrows represent coarse-to-fine refinement.

position relationships among its elements and is regarded as representing the

plant's constituents at a certain spatial resolution, we call it a *scale*.

A multiscale model can potentially transcend the scope of a single plant. For example, the focus of a modeller may be fixated on a population of plants without disregarding structures of individual plants or even molecular processes. In such scenarios, the reference to incomparable modularities in individual plants, as perceived from the population scale, necessitates the aggregation of finer modularities into common coarse representations (Figure 4.5B). The dissolution of coarse modularities to common fine representations (Figure 4.5C) may occur when microscopic models are taken into account. Furthermore, arbitrary measurements or data in three dimensional space compartments (e.g. layers or boxes) that cannot be classified as plant modularities can thus be easily included in the refinement ordering as properties referring to specific scales. The distinction between scales and modularities is intentional to highlight scales as sets of modularities as well as the transcendence beyond the scope of single organisms. If incomparable scales exist, a linear ordering is unsuitable for representation. In such cases, a generalized multiscale graph [67] or a structure-of-scales [128] can represent the scales as a partially ordered set. Such refinement orderings (Figure 4.6A) are useful for models with an extensive range of scales (as we will demonstrate in Part III of this thesis).

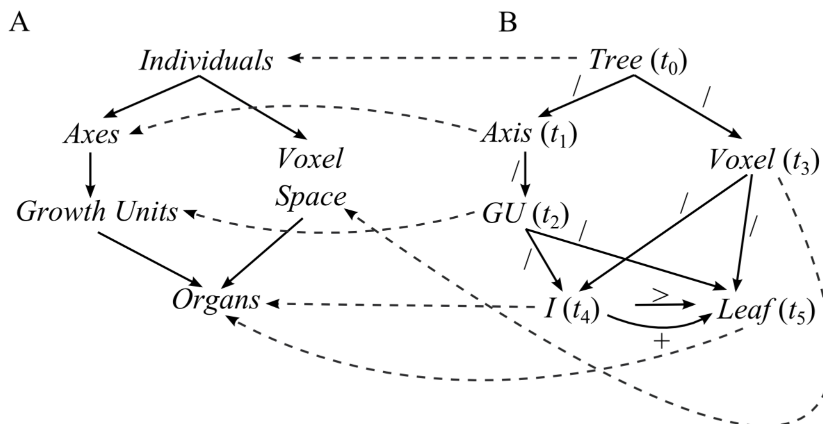


Figure 4.6: Structure-of-Scales $(\Lambda_s, \leq)$ and Type Graph T. A: graph representation of partially-ordered set of scales. Edges labels are not shown since all edges are labelled /. B: type graph T corresponding to the structure-of-scales. Dotted arrows represent the scale mapping T_s from T_v to Λ_s.

Definition 4.3.1 (**Structure-of-Scales**). *A* structure-of-scales *[128] is a finite weak partially ordered set* $(\Lambda_s, \leq)$. *The elements of* Λ_s *are called scale labels,* s_i. *If* $\Lambda_s \ni s_1 \leq s_2 \in \Lambda_s$, *we say that* s_1 *is finer than* s_2 *and that* s_2 *is coarser than* s_1. $inf(s_1, s_2) \in \Lambda_s$ *is the coarsest common refinement of* s_1 *and* s_2. $sup(s_1, s_2) \in \Lambda_s$ *is the finest common coarsening of* s_1 *and* s_2. *Let* $\prec$ *be the nearest neighbour relation of* $(\Lambda_s, \leq)$, *i.e.,* $s_1 \prec s_2 :\Leftrightarrow s_1 \leq s_2 \wedge s_1 \neq s_2 \wedge \forall s \in \Lambda_s : (s_1 \leq s \leq s_2 \Rightarrow s = s_1 \vee s = s_2)$. *Then we can characterize* $(\Lambda_s, \leq)$ *by the directed acyclic graph* $G_s = (\Lambda_s, \{(s_1, /, s_2) | s_1, s_2 \in \Lambda_s, s_1 \succ s_2\}, G_{s\lambda} : s_i \mapsto s_i)$ *over the alphabet* $(\Lambda_s, \{/\})$, *where the vertex set is also the set of node labels. Edges are directed only from coarser to finer scales.*

Example 4.3.1 (**Structure-of-Scales**). *An example of a structure-of-scales* $(\Lambda_s, \leq)$ *is shown in Figure 4.6A. In this case,* individuals, axes, voxel space, growth units, organs *are scale labels, i.e. elements of the set* Λ_s. *We say, for example, that* growth units *are finer than* individuals, *or* individuals *are coarser than* growth units, *written* growth units $\leq$ individuals. *The coarsest common refinement of* voxel space *and* axes *are* organs, *written* inf(voxel space, axes) = organs. *The finest common coarsening of* voxel space *and* growth units *are* individuals, *written* sup(voxel space, growth units) = individuals. *The nearest neighbour relations and their corresponding edges in the graph representation of* $(\Lambda_s, \leq)$, G_s *are:*

individuals $\succ$ axes	(individuals, /, axes)
individuals $\succ$ voxel space	(individuals, /, voxel space)
axes $\succ$ growth units	(axes, /, growth units)
voxel space $\succ$ organs	(voxel space, /, organs)
growth units $\succ$ organs	(growth units, /, organs)

4.3.2 Type Graph

Graph re-writing and rule-based operations are generally defined for *types* rather than *scales*. For example, one scale may have leaf and internode types, while another, coarser scale may have an axis type. In order to provide control on the possible relationships between types, we adopt a typed approach from languages such as Java (in which types are called classes) and require a specification of all types and their potential relationships in a *type graph* ([91], [128]) before transformation rules are established.

Definition 4.3.2 (**Type Graph**). *Let $\Lambda = (\Lambda_v, \Lambda_e, (\Lambda_s, \leq))$ be an alphabet consisting of node labels Λ_v, edge labels Λ_e and a structure of scales $(\Lambda_s, \leq)$. A* type graph *[128] $T = (T_v, T_e, T_\lambda, T_s)$ is a labelled, directed graph over Λ consisting of a set of vertices T_v, a set of labelled edges $T_e \subseteq T_v \times \Lambda_e \times T_v$, a surjective node labelling function $T_\lambda : T_v \to \Lambda_v$ and a surjective scale mapping function $T_s : T_v \to \Lambda_s$ fulfilling the condition $\forall u, v \in T_v : (T_\lambda(u) = T_\lambda(v) \Rightarrow T_s(u) \neq T_s(v))$. T is not necessarily acyclic and has a distinguished edge label $/ \in \Lambda_e$ ("refinement") which is antitonic with the scale mapping, i.e., $(u, /, v) \in T_e \Rightarrow T_s(u) > T_s(v)$.*

One can thus say that the refinement relationships between the types are controlled or restricted by the structure-of-scales based on the scale mapping function. In other words, the graph representation of the structure-of-scales is an epimorphic (see Sections 3.1.4 and 3.1.2.3 on epimorphism) image of the type graph.

Each type is represented by a node in the type graph. Refinement relationships between types are represented by edges labelled by the unique refinement edge label. The appearance of other edges, such as successor and branching edges, between two nodes a and b in a type graph means that edges with these labels are allowed between instances of types a and b, although it is not necessary that they exist. We discuss *instances* of *types* in the next Section 4.3.3.

Example 4.3.2 (**Type Graph**). *We give an example of a* type graph *following the structure-of-scales $(\Lambda_s, \leq)$ defined in Example 4.3.1. Let $\Lambda = (\Lambda_v, \Lambda_e, (\Lambda_s, \leq))$ be an alphabet consisting of node labels $\Lambda_v = \{Tree, Axis, GU, Voxel, I, Leaf\}$, edge labels $\Lambda_e = \{>, +, /\}$ and a structure of scales $(\Lambda_s, \leq)$. GU is short for growth unit and I is short for internode. We specify a type graph $T = (T_v = \{t_0, t_1, t_2, t_3, t_4, t_5\}, T_e, T_\lambda, T_s)$. T_e consists of the edges $(t_0, /, t_1)$, $(t_0, /, t_3)$, $(t_1, /, t_2)$, $(t_2, /, t_4)$, $(t_2, /, t_5)$, $(t_3, /, t_4)$, $(t_3, /, t_5)$, $(t_4, >, t_5)$, and $(t_4, +, t_5)$. The nodes are labelled $T_\lambda(t_0) = Tree$, $T_\lambda(t_1) = Axis$, $T_\lambda(t_3) = Voxel$, $T_\lambda(t_2) = GU$, $T_\lambda(t_4) = I$, and $T_\lambda(t_5) = Leaf$. A graphical image of T is shown in Figure 4.6B. The individual types represented as nodes in T are mapped to scales as $T_s(t_0) = Individuals$, $T_s(t_1) = Axes$, $T_s(t_3) = Voxel\ Space$, $T_s(t_2) = Growth\ Units$, $T_s(t_3) = T_s(t_5) = Organs$. The scale mappings are shown as dotted arrows in Figure 4.6. In this example, the $Organs$ scale consists of two types: I and $Leaf$, while the rest of the scales have a single representative type each.*

4.3.3 Instanced Graph

Analogous to the creation of object instances from classes in the object-oriented programming paradigm, instances of types can be created as nodes in an *instanced graph*. This graph is used for modelling virtual entities such as biological organisms, cells, forest stands, etc., which are represented as nodes, and their relationships as edges.

The *type graph* is a homomorphic image of the *instanced graph*, with the underlying homomorphism preserving node types and edge labels. This means that the type graph controls or restricts the architecture of the instanced graph. If two types a and b are, for example, not connected by a successor edge in the type graph, such an edge is forbidden between instances of a and b.

Definition 4.3.3 (**Instanced Graph**). *An* instanced graph *(also known as multi-scale typed graph in [128])* $G = (G_v, G_e, G_\lambda, G_t)$ *over a type graph* T *is a graph* (G_v, G_e, G_λ) *with a graph homomorphism* $G_t : (G_v, G_e, G_\lambda) \to T$*.* G_λ *is defined by taking labels from* T *such that* $G_\lambda(v) = T_\lambda(G_t(v))$*.* G *is not necessarily acyclic.*

The *instanced graph* is the subject of rule-based transformations. Successor and branching edges in *instanced graphs* are interpreted in the same way as the conventional successor and branching relations in L-systems.

Example 4.3.3 (**Instanced Graph**). *We give an example of an* instanced graph G *following the structure-of-scales* $(\Lambda_s, \leq)$ *and type graph* T *defined in Example 4.3.2. Figure 4.7 shows the instanced graph. It should be clear from the figure that node labels, i.e.* $Tree$*,* $Axis$*,* GU*,* $Voxel$*,* I*,* $Leaf$*, indicate the homomorphism to the type graph* T*.*

Together, the *structure-of-scales* (Definition 4.3.1), the *type graph* (Definition 4.3.2), and the *instanced graph* (Definition 4.3.3) form a three-part multi-scale graph data structure fundamental to the concepts we introduce in the rest of this chapter.

4.4 Multiscale Graph Rewriting

This section describes modifications of the L-system-style connection mechanism illustrated in Section 3.3.2.3. For clarity, the explanations are given

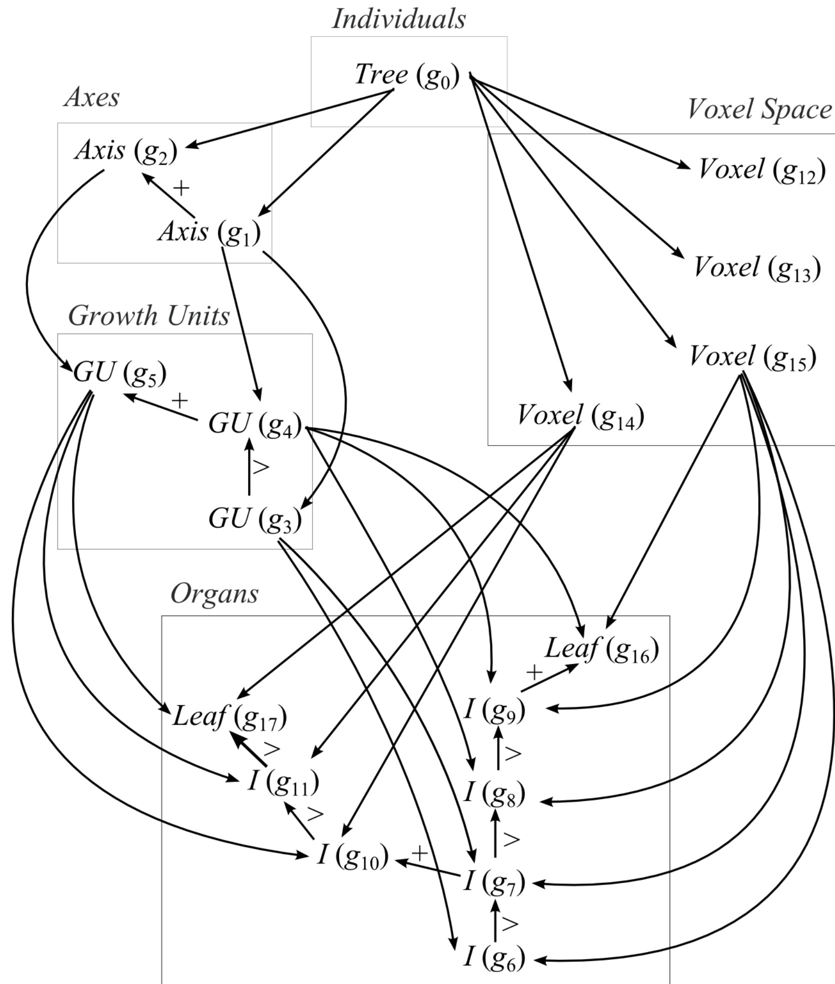


Figure 4.7: Instanced Graph corresponding the structure-of-scales and type graph shown in Figure 4.6. Nodes in the same scale are boxed together. To avoid cluttering, only non-refinement edges, i.e. successor (>)and branching (+) edges are labelled. Edges without labels are refinement edges.

progressively, with Section 4.4.2 building upon Section 4.4.1. The modifications result in a connection mechanism suitable for rewriting multi-scale *instanced graphs* (defined in Section 4.3.3). This mechanism addresses the problem stated earlier in Section 4.2 and tackles some additional issues that we will describe next. For simplicity, we refer to *instanced graph* whenever

we use the term *graph* in this section.

4.4.1 Scale-specific L-system-style Connection

The L-system-style connection mechanism does not differentiate nodes at different scales. We tackle this immediately by incorporating scale discrimination in the mechanism to get the so-called *scale-specific L-system-style connection*.

Let Λ_s be an alphabet of scale labels, $(\Lambda_s, \leq)$ be a structure-of-scales, $\Lambda = (\Lambda_v, \Lambda_e, (\Lambda_s, \leq))$ be an alphabet, and T be a type graph over $(\Lambda_s, \leq)$. The refinement edge label $/$, the successor edge label $>$, and the branching edge label $+$ are elements in Λ_e. A production p using the *scale-specific L-system-style connection* is $p : L \xrightarrow{\sigma,\tau} R$ over $(\Lambda_s, \leq)$ and T, where σ and τ are sets of connection instructions. The left-hand side L of the production is specified as $L = (L_G, L_L, L_R)$ and the right-hand side R of the production is specified as $R = (R_G, R_L, R_R, R_B)$. We describe these constituents of L and R as follows:

- L_G and R_G are graphs, i.e. $L_G = (L_{Gv}, L_{Ge}, L_{G\lambda}, L_{Gt})$ and $R_G = (R_{Gv}, R_{Ge}, R_{G\lambda}, R_{Gt})$. L_G is used as the graph for matching in a host graph G on which the production is applied. R_G on the other hand is the graph produced for each match.

- As in the case of L-system-style connections, L_L and R_L are the sets of left-most nodes in L_G and R_G respectively. More specifically, $L_L \subseteq L_{Gv}$ and $R_L \subseteq R_{Gv}$.

- L_R and R_R are the sets of right-most nodes in L_G and R_G respectively, i.e. $L_R \subseteq L_{Gv}$ and $R_R \subseteq R_{Gv}$.

- The last constituent $R_B \subseteq R_{Gv}$ is the set of pending branch nodes in R_G.

Hence, in a longer form, the production p can be written as

$$p : ((L_{Gv}, L_{Ge}, L_{G\lambda}, L_{Gt}), L_L, L_R) \xrightarrow{\sigma,\tau} ((R_{Gv}, R_{Ge}, R_{G\lambda}, R_{Gt}), R_L, R_R, R_B).$$

For the sets of connection instructions σ and τ, we exclude the refinement edge label (symbolized as /) from the operation of the operators N_e^{dir}, where *dir* is edge direction and e is edge label. They are specified as follows (compare with the connection instructions in Section 3.3.2.2):

$$\sigma = \bigcup_{/\neq\gamma\in\Lambda_E, s\in R_R\setminus\{\%\}} \{(N_\gamma^{out},\gamma,s),(N_\gamma^{out},+,s)\}$$
$$\tau = \bigcup_{/\neq\gamma\in\Lambda_E, t\in R_L\setminus\{\%\}} \{(N_\gamma^{in},\gamma,t)\} \cup \bigcup_{/\neq\gamma\in\Lambda_E, t\in R_B} \{(N_\gamma^{in},+,t)\} \ .$$

With the above modifications, p operates like the L-system-style connection mechanism (Section 3.3.2.3) with one exception: an embedding edge (s,γ,t) or $(s,+,t)$ is only established if s and t are at the same scale. More specifically, all embedding edges (s,γ,t) or $(s,+,t)$ in the derivation $G \xrightarrow{p} H$ must satisfy $T_s(H_t(s)) = T_s(H_t(t))$, where $s \in H_v$ and $t \in H_v$. We call this the *scale condition*. To recap, H_t is the homomorphism from H to the type graph T, and T_s is the scale mapping function from T to the structure-of-scales $(\Lambda_s, \leq)$.

Note that we have yet to explain how the left-most, right-most, and pending branch nodes (i.e. L_L, L_R, R_L, R_R, and R_B) at multiple scales have been derived. This will be left to Section 4.5, where syntax changes which play a more important role in these derivations are documented. Nevertheless, an example of this connection mechanism can already be presented.

Example 4.4.1 (**Scale-specific L-system-style Connection**). *Let* $\Lambda_s = \{a,b\}$ *be an alphabet of scale labels and* $\Lambda = (\Lambda_v = \{A,B\}, \Lambda_e = \{>,+,/\}, (\Lambda_s,\leq))$ *be an alphabet, where* $(\Lambda_s,\leq)$ *is a structure-of-scales such that* $b < a$*. A type graph* $T = (T_v = \{t_0,t_1\},\ T_e = \{(t_0,/,t_1)\}, T_\lambda, T_s)$ *over* Λ *is given such that* $T_\lambda(t_0) = A$*,* $T_\lambda(t_1) = B$*,* $T_s(t_0) = a$*, and* $T_s(t_1) = b$*. A host graph (an instanced graph)* $G = (G_v = \{g_0,g_1,g_2,g_3\},\ G_e = \{(g_0,/,g_1),(g_2,/,g_3),(g_0,>,g_2),(g_1,>,g_3)\}, G_\lambda, G_t)$ *is specified.* G_t *is a homomorphism* $G_t : G \to T$ *such that* $G_t(g_0) = G_t(g_2) = t_0$ *and* $G_t(g_1) = G_t(g_3) = t_1$*.* G_λ *labels the nodes based on* G_t*, i.e.* $G_\lambda(g_0) = G_\lambda(g_2) = T_\lambda(G_t(g_0)) = T_\lambda(G_t(g_2)) = A$ *and* $G_\lambda(g_1) = G_\lambda(g_3) = T_\lambda(G_t(g_1)) = T_\lambda(G_t(g_3)) = B$*. The structure-of-scales* $(\Lambda_s,\leq)$*, type graph* T*, and graph* G *are shown in Figure 4.8.*

We specify a scale-specific L-system-style connection *production* $p : L \xrightarrow{\sigma,\tau} R$*, where* $L = (L_G, L_L, L_R)$ *and* $R = (R_G, R_L, R_R, R_B)$*.* L_G *is graph* $(\{l_0, l_1\},$

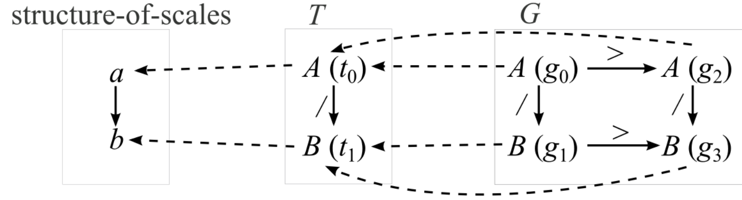


Figure 4.8: Scale specific L-system-style connection setup. Left: structure-of-scales. Middle: type graph T. Right: host graph G. Dotted arrows from T to the structure-of-scales represent the scale mapping function T_s. Dotted arrows from G to T represent the graph homomorphism G_t (edge mappings not shown).

$\{(l_0, /, l_1)\}, L_{G\lambda}, L_{Gt})$ and R_G is a graph $(\{r_0, r_1, r_2, r_3\}, \{(r_0, /, r_1), (r_2, /, r_3), (r_0, >, r_2), (r_1, >, r_3)\}, R_{G\lambda}, R_{Gt})$. The homomorphism from L_G to T is given as $L_{Gt}(l_0) = t_0$ and $L_{Gt}(l_1) = t_1$. The labelling of nodes in L_G is based on L_{Gt}, such that $L_{G\lambda}(l_0) = T_\lambda(L_{Gt}(l_0)) = A$ and $L_{G\lambda}(l_1) = T_\lambda(L_{Gt}(l_1)) = B$. The homomorphism from R_G to T is given as $R_{Gt}(r_0) = R_{Gt}(r_2) = t_0$ and $R_{Gt}(r_1) = R_{Gt}(r_3) = t_1$. The labelling of nodes in R_G is based on R_{Gt}, such that $R_{G\lambda}(r_0) = T_\lambda(R_{Gt}(r_0)) = R_{G\lambda}(r_2) = T_\lambda(R_{Gt}(r_2)) = A$ and $R_{G\lambda}(r_1) = T_\lambda(R_{Gt}(r_1)) = R_{G\lambda}(r_3) = T_\lambda(R_{Gt}(r_3)) = B$. The set of left-most nodes L_L in L_G is $\{l_0, l_1\}$ and the set of right-most nodes L_R in L_G is also $\{l_0, l_1\}$. The set of left-most nodes R_L in R_G is $\{r_0, r_1\}$ while the set of right-most nodes R_R in R_G is $\{r_2, r_3\}$. R_B is an empty set in this example. Figure 4.9 shows production p in graphical form.

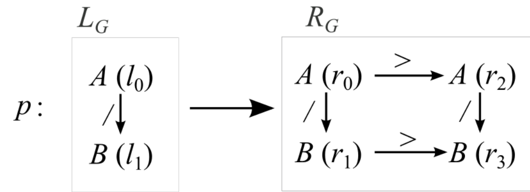


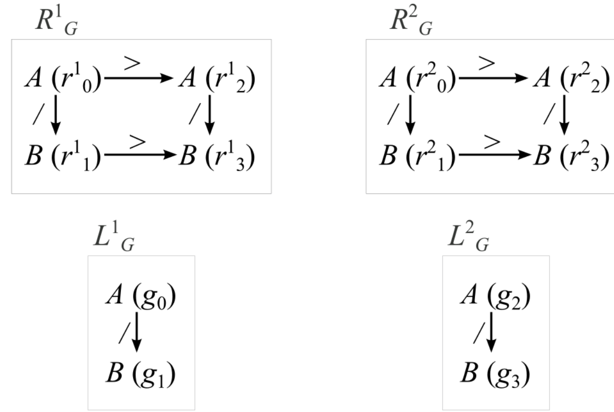
Figure 4.9: Scale specific L-system-style production. The sets of left-most (L_L) and right-most (L_R) nodes of the left-hand side graph L_G are both $\{l_0, l_1\}$. The sets of left-most (R_L) and right-most (R_R) nodes of the right-hand side graph R_G are $\{r_0, r_1\}$ and $\{r_2, r_3\}$ respectively.

We now apply p on G to derive graph H. The derivation follows the five-

step procedure for L-system-style connections in Section 3.3.2.3 with modifications as described above.

Step 1*:*

For each match m^i *(*$1 \leq i \leq n$*,* n *being the number of matches), an isomorphic copy* R^i *of* R *is derived using the sequential SPO method described in Section 3.2.2. In this example, there is a match on* $g_0 \xrightarrow{/} g_1$ *and another on* $g_2 \xrightarrow{/} g_3$*, resulting in the following* R^i_G *graphs (corresponding matched subgraphs* L^i_G *are shown below each* R^i_G*:*



Step 2*:*

The embedding edges are determined following the operator-based approach in Section 3.3.2.2. For the match m^1*, the half-edges*

$$e_0 = (r^1_2, >, (g_2)?) \qquad e_4 = (r^1_2, +, (g_2)?)$$
$$e_1 = (r^1_2, >, (g_3)?) \qquad e_5 = (r^1_2, +, (g_3)?)$$
$$e_2 = (r^1_3, >, (g_2)?) \qquad e_6 = (r^1_3, +, (g_2)?)$$
$$e_3 = (r^1_3, >, (g_3)?) \qquad e_7 = (r^1_3, +, (g_3)?)$$

as pending connections to replacements of g_2 *and* g_3*, i.e.* R^2_G*, are created. For the match* m^2*, the half-edges*

$e_8 = ((g_0)?, >, r^2_0)$ $\quad$ $e_{10} = ((g_0)?, >, r^2_1)$

$e_9 = ((g_1)?, >, r^2_0)$ $\quad$ $e_{11} = ((g_1)?, >, r^2_1)$

as pending connections to replacements of g_0 *and* g_1*, i.e.* R^1_G*, are created. The pairs of complementing half-edges are:*

$p_0 = (e_0, e_8) = (r^1_2, >, r^2_0)$ $\quad$ $p_8 = (e_2, e_8) = (r^1_3, >, r^2_0)$

$p_1 = (e_0, e_9) = (r^1_2, >, r^2_0)$ $\quad$ $p_9 = (e_2, e_9) = (r^1_3, >, r^2_0)$

$p_2 = (e_0, e_{10}) = (r^1_2, >, r^2_1)$ $\quad$ $p_{10} = (e_2, e_{10}) = (r^1_3, >, r^2_1)$

$p_3 = (e_0, e_{11}) = (r^1_2, >, r^2_1)$ $\quad$ $p_{11} = (e_2, e_{11}) = (r^1_3, >, r^2_1)$

$p_4 = (e_1, e_8) = (r^1_2, >, r^2_0)$ $\quad$ $p_{12} = (e_3, e_8) = (r^1_3, >, r^2_0)$

$p_5 = (e_1, e_9) = (r^1_2, >, r^2_0)$ $\quad$ $p_{13} = (e_3, e_9) = (r^1_3, >, r^2_0)$

$p_6 = (e_1, e_{10}) = (r^1_2, >, r^2_1)$ $\quad$ $p_{14} = (e_3, e_{10}) = (r^1_3, >, r^2_1)$

$p_7 = (e_1, e_{11}) = (r^1_2, >, r^2_1)$ $\quad$ $p_{15} = (e_3, e_{11}) = (r^1_3, >, r^2_1)$

where duplicates exist: $p_0 = p_1 = p_4 = p_5$*,* $p_2 = p_3 = p_6 = p_7$*,* $p_8 = p_9 = p_{12} = p_{13}$*,* $p_{10} = p_{11} = p_{14} = p_{15}$*.* p_2*,* p_8 *and their duplicates are rejected because they do not satisfy the* scale condition*. As a result, two embedding edges are synthesized from the connection instructions* σ *and* τ*. They are* $(r^1_2, >, r^2_0)$ *and* $(r^1_3, >, r^2_1)$*.*

Step 3 to Step 5*:*

The remaining steps (step 3 to step 5) of the derivation are the same as in Section 3.3.2.2 and we obtain the derived graph H *shown in Figure 4.10.*

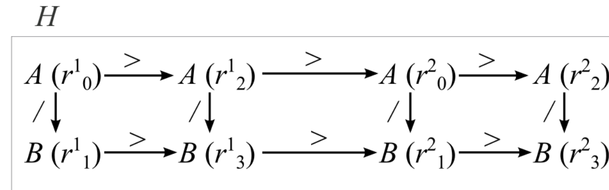
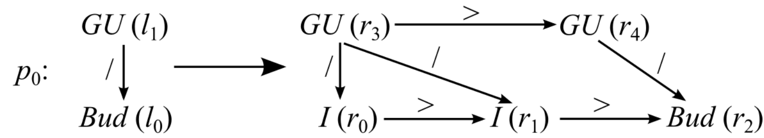


Figure 4.10: Scale specific L-system-style derived graph H. The embedding edges are $(r^1_2, >, r^2_0)$ and $(r^1_3, >, r^2_1)$.

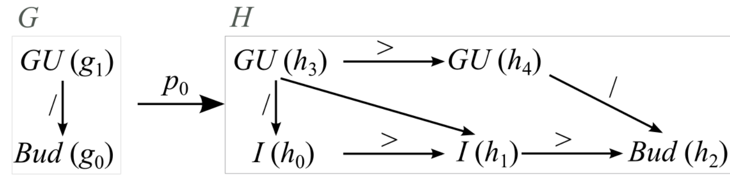
In Example 4.4.1, the *scale-specific L-system-style connection* performs graph rewriting as though there is an L-system rule at each scale, i.e. one rule for scale a and nodes labelled A, and another for scale b and nodes labelled B. Although this is already an enhancement from the original *L-system-style connection*, we have yet to resolve the problem in Section 4.2.

4.4.2 Multiscale Connection

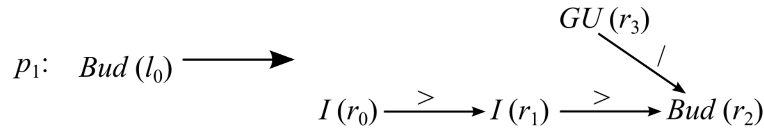
This section describes the *multi-scale connection* method that improves upon the *scale-specific L-system-style connection* method (in Section 4.4.1), which creates embedding edges only for nodes at scales that are explicitly specified in the production. For example, a *scale-specific L-system-style connection* production $p_0 : L \xrightarrow{\sigma,\tau} R$:



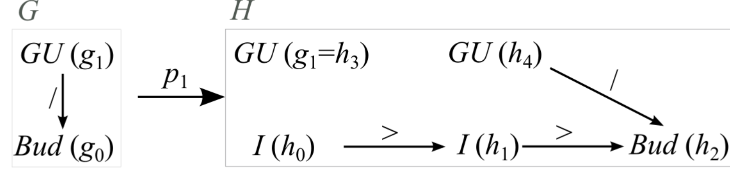
has two scales $Growth\ Units$ and $Organs$ containing the types GU (growth unit), I (internode), and Bud. Here, we omit the structure-of-scales and type graph specifications to keep the description concise. An application of p_0 on a graph G successfully embeds all scales in a derivation as below:



Another similar *scale-specific L-system-style connection* production $p_1 : L \xrightarrow{\sigma,\tau} R$:



fails to embed the *Growth Units* scale when applied on G:



This failure is firstly reminiscent of the problem in Section 4.2 because $GU(g_1 = h_3)$ is not reconnected to $I(h_0)$ and $I(h_1)$ after derivation. Secondly, we would like $GU(g_1 = h_3)$ to have an edge connection to $GU(h_4)$, the newly created GU node. If the *scale-specific L-system-style connection* mechanism is improved to handle these two aspects of the failure, we would be able to use shorter rules like p_1 to achieve the same derivations obtained by using longer rules like p_0. We begin to describe the *multi-scale embedding* method by addressing the latter aspect of the failure.

Recall from Section 4.4.1 that the operator N_e^{dir} in the connection instructions returns nodes connected to (either the left or right-most nodes in) the predecessor graph L^i of graph R^i produced by a match m^i, $1 \leq i \leq n$, where n is the number of matches found. A schematic diagram showing the scope of nodes returned by an operator N_e^{dir} is shown in Figure 4.11. The nodes returned by N_e^{dir} are never connected to nodes in L^i by refinement (/) edges due to their exclusion from the connection instruction sets σ and τ in any *scale-specific L-system-style connection* production $p : L \xrightarrow{\sigma,\tau} R$.

In order to establish embedding edges with un-matched nodes that are encoarsements or refinements of matched nodes, e.g. $GU(g_1 = h_3)$ in the failure above, we require a new operator $S^{bearing}$, which can either be S^l (l stands for left) or S^r (r stands for right). S^l returns nodes connected to the left-most nodes L_L^i while S^r returns nodes connected to the right-most nodes L_R^i of a matched subgraph L^i via paths made up of only refinement (/) edges. With $S^{bearing}$, we gain access to other scales of a matched subgraph without explicitly specifying them in the left-hand side of a production $p : L \xrightarrow{\sigma,\tau} R$. We proceed to include $S^{bearing}$ in the *scale-specific L-system-style connection* to get the *partial multi-scale embedding* mechanism.

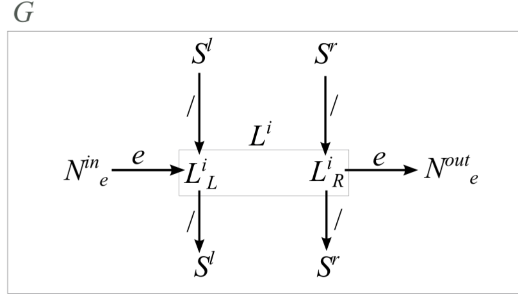


Figure 4.11: Operators for multi-scale embedding. G is the host graph on which a multi-scale embedding production p is applied. L^i in the middle is a matched subgraph of G and it contains the sets of left-most nodes L^i_L and right-most nodes L^i_R. The operator N^{dir}_e for *scale-specific L-system-style connections* returns nodes connected to nodes in L^i_L and L^i_R. The operators S^l and S^r return nodes connected to nodes in L^i_L and L^i_R respectively via directed paths consisting of only refinement edges (labelled /).

4.4.2.1 Partial Multiscale Embedding

Definition 4.4.1 (Partial Multiscale Embedding). *Let Λ_s be an alphabet of scale labels, $(\Lambda_s, \leq)$ be a structure-of-scales, $\Lambda = (\Lambda_v, \Lambda_e, (\Lambda_s, \leq))$ be an alphabet, and T be a type graph over $(\Lambda_s, \leq)$. The refinement edge label /, the successor edge label $>$, and the branching edge label $+$ are elements in Λ_e. Let $p : L \xrightarrow{\sigma,\tau} R$ be a scale-specific L-system-style connection production over $(\Lambda_s, \leq)$ and T. A* partial multi-scale embedding *production is a scale-specific L-system-style connection production $q : L \xrightarrow{\sigma,\upsilon} R$, where*

$$\sigma = \bigcup_{/\neq\gamma\in\Lambda_E, s\in R_R\setminus\{\%\}} \{(N^{out}_\gamma, \gamma, s), (N^{out}_\gamma, +, s)\} \text{ and}$$

$$\upsilon = \bigcup_{/\neq\gamma\in\Lambda_E, t\in R_L\setminus\{\%\}} \{(N^{in}_\gamma, \gamma, t)\} \cup \bigcup_{/\neq\gamma\in\Lambda_E, t\in R_B} \{(N^{in}_\gamma, +, t)\}$$
$$\cup \bigcup_{t\in R_L\setminus\{\%\}} \{(S^r_n, >, t), (S^r_y, +, t)\} \cup \bigcup_{t\in R_B} \{(S^r, +, t)\} .$$

In other words, σ is as defined for p and

$$\upsilon = \tau \cup \bigcup_{t\in R_L\setminus\{\%\}} \{(S^r_n, >, t), (S^r_y, +, t)\} \cup \bigcup_{t\in R_B} \{(S^r, +, t)\}.$$

For a match m^i based on q on a host graph G, let the matched subgraph be

$L^i = (L^i_G, L^i_L, L^i_R)$. S^r is an operator that returns all $w \in G_v$, $w \notin L^i_G$, and there exists a path with only refinement edges between w and $z \in L^i_R$, $w \neq z$. S^r_n is a specific S^r operator that filters its results by excluding nodes that have an existing out-going successor edge connection. S^r_y is a specific S^r operator that filters its results by excluding nodes that do not have an existing out-going successor edge connection. The other, unused operator S^l is an operator that returns all $x \in G_v$, $x \notin L^i_G$, and there exists a path with only refinement edges between x and $y \in L^i_L$, $x \neq y$. The operators S^l and S^r are collectively referred to as $S^{bearing}$. For simplicity, we refer to a partial multi-scale embedding production as partial MS production*.*

Example 4.4.2 (**Partial Multiscale Embedding**)**.** *Let $\Lambda_s = \{GUs, IBs\}$ be an alphabet of scale labels and $\Lambda = (\Lambda_v = \{GU, I, Bud\}, \Lambda_e = \{>, +, /\}, (\Lambda_s, \leq))$ be an alphabet where $(\Lambda_s, \leq)$ is a structure-of-scales such that $IBs \leq GUs$. GUs is short for growth units and IBs is short for internodes and buds. A type graph $T = (T_v = \{t_0, t_1, t_2\}$, $T_e = \{(t_2, /, t_0)$, $(t_2, /, t_1)$, $(t_0, >, t_1)$, $(t_1, >, t_0)$, $(t_0, +, t_1)$, $(t_1, +, t_0)$, $\}, T_\lambda, T_s)$ over Λ is given such $T_\lambda(t_0) = I$, $T_\lambda(t_1) = Bud$, $T_\lambda(t_2) = GU$, $T_s(t_2) = GUs$, and $T_s(t_0) = T_s(t_1) = IBs$.*

A host graph (an instanced graph) $G = (G_v = \{g_0, g_1\}$, $G_e = \{(g_1, /, g_0)\}$, G_λ, $G_t)$ is specified. G_t is a homomorphism $G_t : G \to T$ such that $G_t(g_0) = t_1$ and $G_t(g_1) = t_2$. G_λ labels the nodes based on G_t, i.e. $G_\lambda(g_0) = T_\lambda(G_t(g_0)) = Bud$ and $G_\lambda(g_1) = T_\lambda(G_t(g_1)) = GU$. The structure-of-scales $(\Lambda_s, \leq)$, type graph T, and graph G are shown in Figure 4.12.

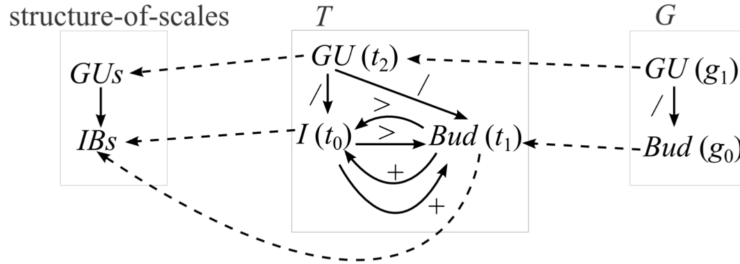


Figure 4.12: Partial multi-scale embedding setup. Left: structure-of-scales $(\Lambda_s, \leq)$. Middle: type graph T. Right: host graph G. Dotted arrows from T to the structure-of-scales represent the scale mapping function T_s. Dotted arrows from G to T represent the graph homomorphism G_t (edge mappings not shown).

We specify a partial multi-scale embedding *production* $q : L \xrightarrow{\sigma,\upsilon} R$*, where*

$L = (L_G, L_L, L_R)$ and $R = (R_G, R_L, R_R, R_B)$. L_G is graph $(\{l_0\}, \emptyset, L_{G\lambda}, L_{Gt})$ and R_G is a graph $(\{r_0, r_1, r_2, r_3\}, \{(r_0, >, r_1), (r_1, >, r_2), (r_3, /, r_2)\}, R_{G\lambda}, R_{Gt})$. The homomorphism from L_G to T is given as $L_{Gt}(l_0) = t_1$. The labelling of nodes in L_G is based on L_{Gt}, such that $L_{G\lambda}(l_0) = T_\lambda(L_{Gt}(l_0)) = Bud$. The homomorphism from R_G to T is given as $R_{Gt}(r_0) = R_{Gt}(r_1) = t_0$, $R_{Gt}(r_2) = t_1$ and $R_{Gt}(r_3) = t_2$. The labelling of nodes in R_G is based on R_{Gt}, such that $R_{G\lambda}(r_0) = T_\lambda(R_{Gt}(r_0)) = R_{G\lambda}(r_1) = T_\lambda(R_{Gt}(r_1)) = I$, $R_{G\lambda}(r_2) = T_\lambda(R_{Gt}(r_1)) = Bud$, and $R_{G\lambda}(r_3) = T_\lambda(R_{Gt}(r_3)) = GU$. The set of left-most nodes L_L in L_G is $\{l_0\}$ and the set of right-most nodes L_R in L_G is also $\{l_0\}$. The set of left-most nodes R_L in R_G is $\{r_0, r_3\}$ while the set of right-most nodes R_R in R_G is $\{r_2, r_3\}$. R_B is an empty set in this example. Figure 4.13 shows production q in graphical form.

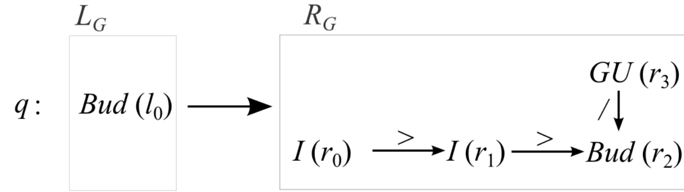
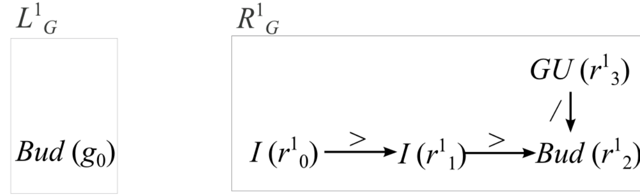


Figure 4.13: Partial multi-scale embedding production q. The sets of left-most (L_L) and right-most (L_R) nodes of the left-hand side graph L_G are both $\{l_0\}$. The sets of left-most (R_L) and right-most (R_R) nodes of the right-hand side graph R_G are $\{r_0, r_3\}$ and $\{r_2, r_3\}$ respectively.

We now apply q on G to derive graph H. The derivation uses the five-step procedure for scale-specific L-system-style connections in Section 4.4.1. In this case, however, there are additional connection instructions as defined in Definition 4.4.1.

Step 1*:*

In this example, m^1 is the one and only match on g_0. An isomorphic copy R^1 of R is derived using the sequential SPO method described in Section 3.2.2, resulting in the following R^1_G graph (corresponding matched subgraph L^1_G is shown on the left of R^1_G):



Step 2:

The embedding edges are determined following the operator-based approach in Section 3.3.2.2. For the match m^1*, the half-edges*

$$e_0 = ((g_1)?, >, r_0^1) \textit{ and } e_1 = ((g_1)?, >, r_3^1)$$

as pending connections to replacements of g_1 *are created.* e_0 *is rejected because it does not satisfy the* scale condition*. As* g_1 *is not matched, the embedding edge* $(g_1, >, r_3^1)$ *is synthesized (for a sequential transformation) from the connection instructions* σ *and* υ*.*

Step 3 to Step 5:

The remaining steps (step 3 to step 5) of the derivation are the same as in Section 4.4.1 (for two-level amalgamation) and we obtain the derived graph H *shown in Figure 4.14.*

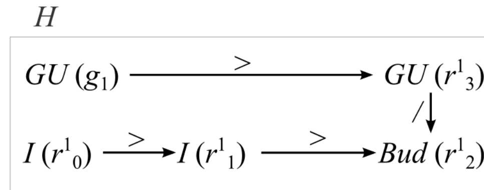


Figure 4.14: Partial multi-scale embedding derived graph H. The embedding edge is $(g_1, >, r_3^1)$.

Example 4.4.2 demonstrates that the *partial multi-scale embedding* (Definition 4.4.1) mechanism establishes embedding edges for scales that are not specified in the left-hand side of the production. The mechanism is demon-

strated again in the more sophisticated Example 4.4.3, where it is subsumed in the (total) *multi-scale embedding* mechanism.

4.4.2.2 Total Multiscale Embedding

In this section, we describe the (total) *multi-scale embedding* method, which, together with the *scale-specific L-system-style connection*, provides a pair of solutions to the graph rewriting problems raised in this chapter.

Definition 4.4.2 (Total Multiscale Embedding). *Let* $q_p : L = (L_G, L_L, L_R) \xrightarrow{\sigma,\upsilon} R = (R_G, R_L, R_R, R_B)$ *be a partial multi-scale production.* L_G *and* R_G *are defined with indexed nodes as* $L_G = (L_{Gv} = \{l_i : 1 \leq i \leq |L_{Gv}|\}, L_{Ge}, L_{G\lambda}, L_{Gt})$ *and* $R_G = (R_{Gv} = \{r_i : 1 \leq i \leq |R_{Gv}|\}, R_{Ge}, R_{G\lambda}, R_{Gt})$ *respectively.*

For an arbitrary relation ϱ *and arbitrary elements* x *and* y*, the* image im *is defined as* $im(x) = \{y|x\varrho y\}$ *and the* preimage pr *is defined as* $pr(x) = \{y|y\varrho x\}$*. By setting* ϱ *as the refinement relationship,* $im(r_i) = \{x \in R_{Gv} : (r_i, /, x) \in R_{Ge}\}$ *and* $pr(r_i) = \{x \in R_{Gv} : (x, /, r_i) \in R_{Ge}\}$*, where* $r_i \in R_{Gv}$*.*

A total multi-scale embedding production q *(*MS production *in short) is an extension of* q_p*,* $q : L \xrightarrow{\eta,\vartheta} R = (R_G, R_L, R_R, R_B, R_C = R_{Cl} \cup R_{Cr}, R_F = R_{Fl} \cup R_{Fr})$*.* R_C *is the set of nodes in* R_{Gv} *with no encoarsements, defined as* $R_C = \{r_i \in R_{Gv} : pr(r_i) = \emptyset\}$*.* R_F *is the set of nodes in* R_{Gv} *with no refinements, defined as* $R_F = \{r_i \in R_{Gv} : im(r_i) = \emptyset\}$*.*

The sets of nodes R_C *and* R_F *each consist of two subsets of nodes, i.e.* $R_C = R_{Cl} \cup R_{Cr}$ *and* $R_F = R_{Fl} \cup R_{Fr}$*. These subsets are defined as* $R_{Cl} = R_C - R_R$*,* $R_{Cr} = R_C \cap R_R$*,* $R_{Fl} = R_F - R_R$*, and* $R_{Fr} = R_F \cap R_R$*. In other words, the nodes in* R_C *and* R_F *are grouped further based on their membership in* R_R*.*

The total multi-scale embedding production $q : L \xrightarrow{\eta,\vartheta} R$ *has the sets of connection instructions* η *and* ϑ *as follows:*

$$\eta = \sigma \cup \bigcup_{s \in R_{Fl}, T_s(R_{Gt}(s)) \succ T_s(G_t(S^l))} \{(s, /, S^l)\} \cup \bigcup_{s \in R_{Fr}, T_s(R_{Gt}(s)) \succ T_s(G_t(S^r))} \{(s, /, S^r)\},$$

$$\vartheta = \upsilon \cup \bigcup_{t \in R_{Cl}, T_s(R_{Gt}(t)) \prec T_s(G_t(S^l))} \{(S^l, /, t)\}$$

$$\cup \bigcup_{t \in R_{Cr}, T_s(R_{Gt}(t)) \prec T_s(G_t(S^r))} \{(S^r, /, t)\}$$

For simplicity, we refer to a total multi-scale embedding production as MS *production.*

Example 4.4.3 (**Total Multiscale Embedding**). *Let $\Lambda_s = \{Axes, GUs, IBs\}$ be an alphabet of scale labels and $\Lambda = (\Lambda_v = \{Axis, GU, Bud, I\}$ $, \Lambda_e = \{>, +, /\}$, $(\Lambda_s, \leq))$ be an alphabet where $(\Lambda_s, \leq)$ is a structure-of-scales such that $IBs \leq GUs \leq Axes$. GUs, GU and I abbreviate* growth units, growth unit *and* internode *respectively. IBs is abbreviation for* internodes *and* buds. *A type graph $T = (T_v = \{t_0, t_1, t_2, t_3\}$, $T_e, T_\lambda, T_s)$ over Λ is given such $T_\lambda(t_0) = Axis$, $T_\lambda(t_1) = GU$, $T_\lambda(t_2) = I$, $T_\lambda(t_3) = Bud$, and $T_s(t_0) = Axes$, $T_s(t_1) = GUs$, $T_s(t_2) = T_s(t_3) = IBs$. T_e is a set of edges $\{(t_0, /, t_1)$, $(t_1, /, t_2)$, $(t_1, /, t_3)$, $(t_2, >, t_3)$, $(t_2, +, t_3)$, $(t_3, >, t_2)$, $(t_3, +, t_2)\}$.*

A host graph (an instanced graph) $G = (G_v = \{g_0, g_1, g_2, g_3\}$, $G_e = \{(g_1, /, g_0)$, $(g_1, /, g_2), (g_1, /, g_3), (g_2, >, g_3)\}$, G_λ, $G_t)$ is specified. G_t is a homomorphism $G_t : G \to T$ such that $G_t(g_0) = t_0$, $G_t(g_1) = t_1$, $G_t(g_2) = t_2$, and $G_t(g_3) = t_4$. G_λ labels the nodes based on G_t, i.e. $G_\lambda(g_0) = T_\lambda(G_t(g_0)) = Axis$, $G_\lambda(g_1) = T_\lambda(G_t(g_1)) = GU$, $G_\lambda(g_2) = T_\lambda(G_t(g_2)) = I$, and $G_\lambda(g_3) = T_\lambda(G_t(g_3)) = Bud$. The structure-of-scales $(\Lambda_s, \leq)$, type graph T, and graph G are shown in Figure 4.12.

We specify a total multi-scale embedding *production $q : L \xrightarrow{\eta,\vartheta} R$, where $L = (L_G, L_L, L_R)$ and $R = (R_G, R_L, R_R, R_B, R_C, R_F)$. L_G is graph $(\{l_0\}$, $\emptyset, L_{G\lambda}, L_{Gt})$ and R_G is a graph $(\{r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7\}$, $\{(r_0, /, r_2)$, $(r_1, /, r_4)$, $(r_1, /, r_5)$, $(r_2, /, r_6)$, $(r_2, /, r_7)$, $(r_3, >, r_4)$, $(r_4, >, r_5)$, $(r_6, >, r_7)$, $(r_3, +, r_6)$ $\}$, $R_{G\lambda}, R_{Gt})$. The homomorphism from L_G to T is given as $L_{Gt}(l_0) = t_3$. The labelling of nodes in L_G is based on L_{Gt}, such that $L_{G\lambda}(l_0) = T_\lambda(L_{Gt}(l_0)) = Bud$. The homomorphism from R_G to T is given as $R_{Gt}(r_0) = t_0$, $R_{Gt}(r_1) = R_{Gt}(r_2) = t_1$, $R_{Gt}(r_3) = R_{Gt}(r_4) = R_{Gt}(r_6) = t_2$, and $R_{Gt}(r_5) = R_{Gt}(r_7) = t_3$. The labelling of nodes in R_G is based on R_{Gt}, such that $R_{G\lambda}(r_0) = T_\lambda(R_{Gt}(r_0)) = Axis$, $R_{G\lambda}(r_1) = T_\lambda(R_{Gt}(r_1)) = R_{G\lambda}(r_2) = T_\lambda(R_{Gt}(r_2)) = GU$, $R_{G\lambda}(r_3) = T_\lambda(R_{Gt}(r_3)) = R_{G\lambda}(r_4) = T_\lambda(R_{Gt}(r_4)) = R_{G\lambda}(r_6) = T_\lambda(R_{Gt}(r_6)) = I$, and $R_{G\lambda}(r_5) = T_\lambda(R_{Gt}(r_5)) = R_{G\lambda}(r_7) = T_\lambda(R_{Gt}(r_7)) = Bud$. The set of left-most nodes L_L in L_G is $\{l_0\}$ and the set of right-most nodes L_R in L_G is also $\{l_0\}$. The set of left-most nodes R_L in R_G is $\{r_1, r_3\}$ while the set of right-most nodes R_R in R_G is $\{r_1, r_5\}$. $R_B = \{r_0, r_2\}$ is the set of pending branch nodes in R_G. The set of nodes in R_G with no encoarsements is $R_C = R_{Cl} \cup R_{Cr} = \{r_0, r_3\} \cup \{r_1\}$. The set of*

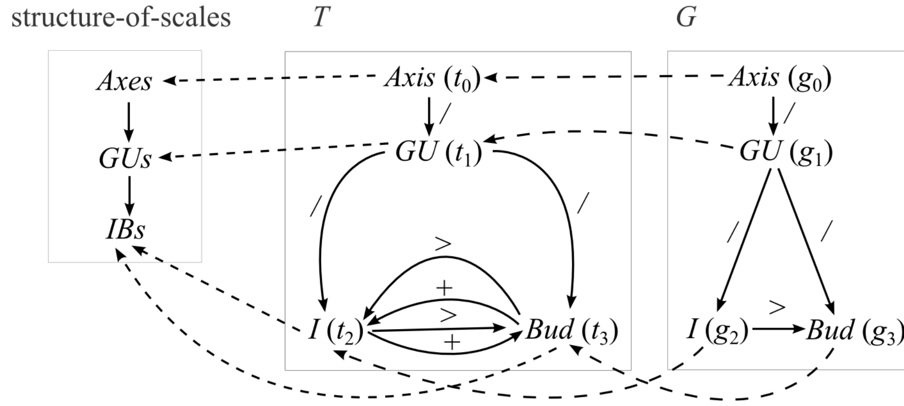


Figure 4.15: Total multi-scale embedding setup. Left: structure-of-scales $(\Lambda_s, \leq)$. Middle: type graph T. Right: host graph G. Dotted arrows from T to the structure-of-scales represent the scale mapping function T_s. Dotted arrows from G to T represent the graph homomorphism G_t (edge mappings not shown).

nodes in R_G with no refinements is $R_F = R_{Fl} \cup R_{Fr} = \{r_3, r_4, r_6, r_7\} \cup \{r_5\}$. Figure 4.16 shows production q in graphical form.

We now apply q on G to derive graph H. The derivation uses the five-step procedure for partial MS productions in Section 4.4.2.1. In this case, however, there are additional connection instructions as defined in Definition 4.4.2.

Step 1*:*

In this example, m^1 is the one and only match on g_3. An isomorphic copy R^1 of R is derived using the sequential SPO method described in Section 3.2.2, resulting in the following R^1_G graph (corresponding matched subgraph L^1_G is shown on the left of R^1_G):

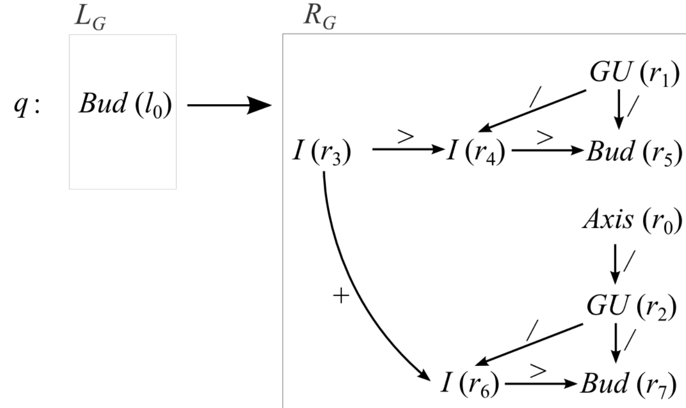
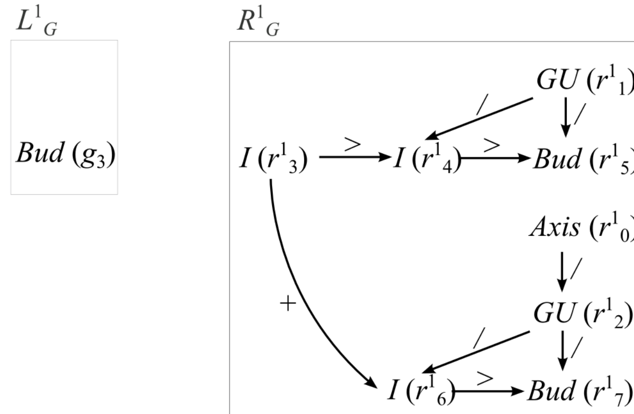


Figure 4.16: Total multi-scale embedding production q. The sets of left-most (L_L) and right-most (L_R) nodes of the left-hand side graph L_G are both $\{l_0\}$. The sets of left-most (R_L) and right-most (R_R) nodes of the right-hand side graph R_G are $\{r_1, r_3\}$ and $\{r_1, r_5\}$ respectively. The set of pending branch nodes R_B in R_G is $\{r_0, r_2\}$. The set of nodes in R_G with no encoarsements is $R_C = R_{Cl} \cup R_{Cr} = \{r_0, r_3\} \cup \{r_1\}$. The set of nodes in R_G with no refinements is $R_F = R_{Fl} \cup R_{Fr} = \{r_3, r_4, r_6, r_7\} \cup \{r_5\}$.



Step 2*:*

The embedding edges are determined following the operator-based approach in Section 3.3.2.2. For match m^1*, the following half-edges are created:*

For connection instructions in the form $(N_\gamma^{in}, \gamma, t)$*:*

$e_0 = ((g_2)?, >, r_1^1) \qquad e_1 = ((g_2)?, >, r_3^1)$

For connection instructions in the form $(N_\gamma^{in}, +, t)$*:*

$e_2 = ((g_2)?, +, r_0^1) \qquad e_3 = ((g_2)?, +, r_2^1)$

For connection instructions in the form $\{(S_n^r, >, t)\}$*:*

$e_4 = ((g_0)?, >, r_1^1) \qquad e_5 = ((g_0)?, >, r_3^1)$

$e_6 = ((g_1)?, >, r_1^1) \qquad e_7 = ((g_1)?, >, r_3^1)$

For connection instructions in the form $\{(S^r, +, t)\}$*:*

$e_8 = ((g_0)?, +, r_0^1) \qquad e_9 = ((g_0)?, +, r_2^1)$

$e_{10} = ((g_1)?, +, r_0^1) \qquad e_{11} = ((g_1)?, +, r_2^1)$

For connection instructions in the form $\{(S^l, /, t)\}$*:*

$$e_{12} = ((g_1)?, /, r_3^1)$$

For connection instructions in the form $\{(S^r, /, t)\}$*:*

$$e_{13} = ((g_0)?, /, r_1^1)$$

e_0*,* e_2*,* e_3*,* e_4*,* e_5*,* e_7*,* e_9 *and* e_{10} *are rejected because they do not satisfy the* scale condition*. As* g_0*,* g_1 *and* g_2 *are not matched, the remaining half edges (*e_1*,* e_6*,* e_8*,* e_{11}*,* e_{12}*,* e_{13}*) are synthesized (based on sequential transformations) from the connection instructions* η *and* ϑ*.*

Step 3 to Step 5*:*

The remaining steps (step 3 to step 5) of the derivation are the same as in Section 4.4.1 (for two-level amalgamation) and we obtain the derived graph H *shown in Figure 4.17.*

Example 4.4.3 demonstrates the solution to the problems stated in Section 4.2. Firstly, refinement edges are established automatically from the remaining host graph to the produced graphs, unlike the operation of single-scale rules shown in Figure 4.3. For example, a refinement edge is established from $Axis(g_0)$ to $GU(r_1^1)$. Secondly, lengthy rules (see rule in Section 4.2 for the production shown in Figure 4.4) to manually establish embedding refinement edges can be avoided. In Example 4.4.3, the MS production q is specified in XL as:

```
Bud ==> I [Axis GU I Bud] GU I Bud;.
```

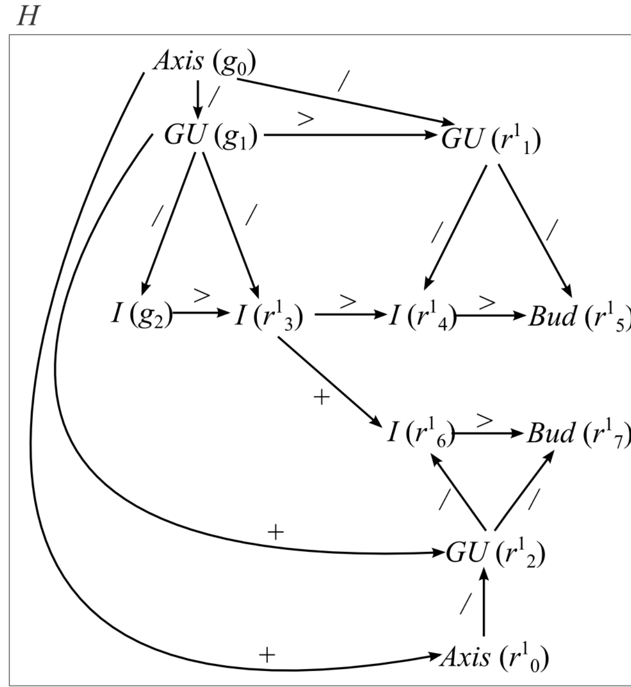


Figure 4.17: Total multi-scale embedding derived graph H.

Notice that no syntax is required to established the embedding edges at all scales. The problems in Section 4.2 are hence resolved.

4.4.2.3 On Parallelism in Multiscale Embedding

Parallel rewriting is supported by MS productions based on the two-level amalgamation mechanism described in Section 4.4.1. However, there are two cases in which parallel rewriting is not supported:

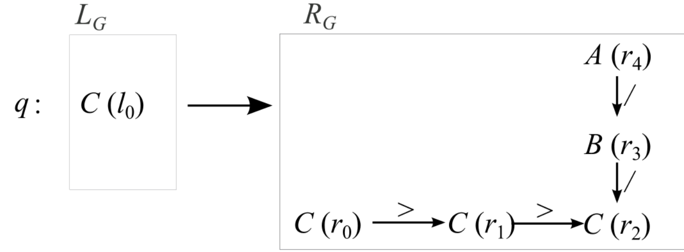
- Matches that overlap. More specifically, given two matches m_1 and m_2 with matched graphs $L^1_G = (L^1_{Gv}, L^1_{Ge}, L^1_{G\lambda}, L^1_{Gt})$ and $L^2_G = (L^2_{Gv}, L^2_{Ge}, L^2_{G\lambda}, L^2_{Gt})$ such that $L^1_{Gv} \cap L^2_{Gv} \neq \emptyset$, rewriting is performed only for the first match (either m_1 or m_2) found. This behaviour is consistent with single-scale rewriting defined in [91].

- Matches with nodes returned by the $S^{bearing}$ operator that are adjacent. More specifically, let m_1 and m_2 be two matches with matched graphs

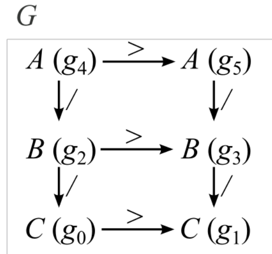
$L_G^1 = (L_{Gv}^1, L_{Ge}^1, L_{G\lambda}^1, L_{Gt}^1)$ and $L_G^2 = (L_{Gv}^2,\ L_{Ge}^2\ , L_{G\lambda}^2\ , L_{Gt}^2)$ such that $L_{Gv}^1 \cap L_{Gv}^2 = \emptyset$. If there exists an edge $(x, e, y) \in G_e$ in the host graph G where $x \in S^{bearing}$ is operating on L_G^1 and $y \in S^{bearing}$ is operating on L_G^2, or $x \in S^{bearing}$ is operating on L_G^2 and $y \in S^{bearing}$ is operating on L_{Gv}^1, embedding edges are not established to and from the nodes returned by the $S^{bearing}$ operator.

We illustrate the argument for the second case (matches adjacent with one another) by considering several scenarios.

Consider a partial MS production $q : L \xrightarrow{\sigma,\upsilon} R$:



where types A, B, and C belong to three distinct scales from coarse to fine. We omit the structure-of-scales and type graph specifications to keep this example concise with focus on the argument against parallelism for adjacent matches. q is applied on a host graph G depicted below:



The derivation $G \xrightarrow{q} H$ results in the graph H shown in Figure 4.18.

q as a partial MS production did not establish embedding edges $(g_6, /, g_0)$, $(g_6, /, g_1)$, $(g_8, /, g_3)$, and $(g_8, /, g_4)$. This was addressed in Section 4.4.2.2 with total multiscale embedding but we now focus on another obvious problem shown in Figure 4.18. For parallel rewriting of adjacent matches with

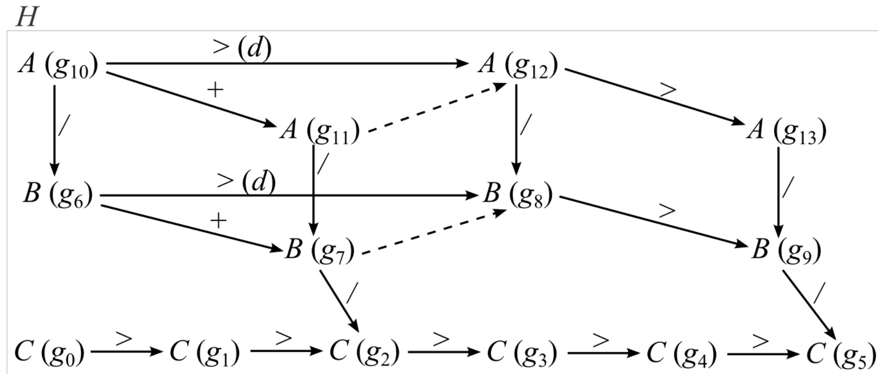
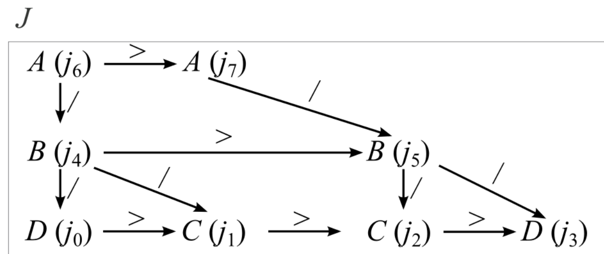


Figure 4.18: Prevention of embedding edges at un-matched scales between adjacent matches during parallel rewriting.

multiple scales to work, i.e. for the dotted edges in Figure 4.18 to be established, the set of connection instructions σ defined for partial MS productions in Section 4.4.2.1 has to be improved by including instructions that utilize the S^l operator. Such an improvement will result in matching half-edges between the newly included instructions using the S^l operator on the $C(g_1)$ match and the instructions using the N_λ^{out} operator on the $C(g_0)$ match, forming the dotted edges in Figure 4.18. Ideally, the (un-matched) edges marked with (d), i.e. edges connected to nodes returned by the operator $S^{bearing}$, should then be automatically deleted to avoid forming circles in the derived graph.

The S^l operator is however left out from the definition of the partial MS production because the enforcement of such edge deletions for adjacent matches at the un-matched scales will cause other conflicting scenarios. For example, consider another host graph J below:

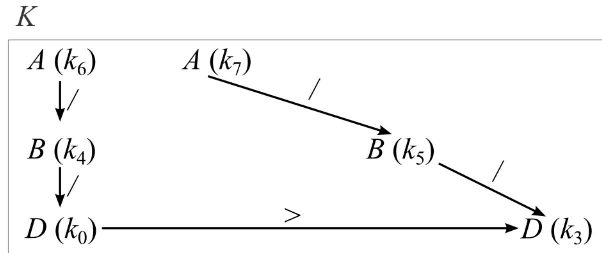


A partial MS production q_2 is applied on J. q_2 is graphically represented as below:

L_G R_G

q_2 : $C\ (l_0)$ ⟶

By deleting edges between nodes returned by the operator $S^{bearing}$, e.g $(j_6, >, j_7)$, we will obtain the derived graph K in the derivation $J \xrightarrow{q_2} K$ shown below:



This result does not satisfy the intent to remove the C nodes, reconnect the pair of D nodes, and keep the relations or edges in the coarser scales containing A and B nodes. Consequently, returning to the earlier derivation $G \xrightarrow{q} H$, we note that unconditional deletion of the (d)-marked edges in Figure 4.18 is not a solution for parallel multi-scale embedding. One can argue that conditional deletion can be introduced, for example, by performing deletion only if the right-hand side of a production like q contains nodes (e.g. $A(r_4)$ above) at the same scales as the nodes for which edge connections are deleted (e.g. $A(g_{10})$ and $A(g_{12})$ above). We argue that this, however, is counter-intuitive from the practical perspective because production rules for edge deletions commonly require the involved nodes to be specified on the left-hand side. To avoid confusion in the implementation language XL, we choose not to perform such hidden deletions. This results in the decision to leave the S^l operator-based instructions out from the definition of the partial MS production in order to avoid the creation of circular structures at the same scale. This is further supported by the fact that the scene graph interpretation for 3D rendering in GroIMP considers circular structures involving

only > and + edges as errors.

In conclusion, parallelism is supported in *multi-scale embedding* except for two special cases. Despite the loss of support for these special cases, the examples in Chapter 6 do not reveal any insufficiencies in terms of parallel and multiscale graph rewriting using the techniques presented in this section.

4.5 XL Multiscale Syntax & Features

The graph model and grammar introduced in this chapter are implemented as an extension to the eXtended L-system (XL) language [91, 128] in the open source software GroIMP [93]. The language is extensive and the reader is encouraged to see [91] and [75] for details of the language. In the following, we highlight fundamentals of the language as well as modified aspects that empower it for multiscale modelling.

4.5.1 Syntax Extensions

In XL, node labels are declared as *modules* with object-orientated functionalities, similar to classes in the Java programming language. For example, the labels *Axis*, *GU*, *Bud* and *I* in Example 4.4.3 can be declared in XL as shown in Listing 4.1.

```
module Axis(float length, float radius);
module GU(float length, float radius);
module I(super.length, super.diameter) extends F;
module Bud(super.radius) extends Sphere;
```

Listing 4.1: XL Module Declaration

Parameters and their data type for each module are specified in parentheses following the module name. For example, the module *Axis* in Listing 4.1 has parameters *length* and *radius* of the data type float. XL provides classes representing geometrical shapes and turtle commands as possible superclasses of modules. For example, the module I in Listing 4.1 extends the turtle command F. I also inherits the parameters of the super class F by specifying them with a preceeding *super.* instead of the usual data type.

A typical program in XL (after module declarations) begins with an init() method as shown in Listing 4.2.

```
protected void init()
[
        {int x=1;}
        Axiom ==> Bud;
]

public void run()
[
        Bud ==> I Bud;
]
```

Listing 4.2: XL Fundamental Methods

This method initializes the graph model or data structure. Normally, an *Axiom* node exists as the initial node in the graph data structure. XL code in the scope of square brackets [and] is used for writing graph queries, rules, or productions. To use Java code in XL, the code must be written within the scope of curly brackets { and } (e.g. the declaration of the primitive integer x in init() in Listing 4.2). Between square brackets, we can write rules that use the L-system-style embedding (see Section 3.3.2.3) by specifying the graph to be matched (i.e. the query graph) on the left, followed by the symbol ==>, and lastly the production graph on the right. In Listing 4.2, such a rule (symbolized by ==>) is used to rewrite the initial *Axiom* node with a *Bud* node. Alternatively, the symbols ==>> and ::> can be used for parallel SPO embedding rules and imperative code execution without changes to graph topology respectively. The root node of the graph data structure can be accessed by using the symbol ^. Additional methods such as the run() method in Listing 4.2 can be written. Public methods such as run() can be triggered via buttons on the graphical user interface of GroIMP.

Conventionally, the symbol ==> represents single-scale L-system-style connections 3.3.2.3. In order for the program to interpret ==> as a MS production rule, a *type graph* must be constructed. For example, the *type graph* in Example 4.4.3 can be constructed in the init() method as shown in Listing 4.3.

```
protected void init()
[
        //Host graph G initialization
        Axiom ==> Axis /> g:GU /> I Bud </ g;
```

```
        //Type graph initialization
        ==>> ^ /> TypeRoot /> Axis /> GU
                /> {# I Bud #};
]
```

Listing 4.3: XL Host and Type Graph Construction

To specify the type graph, the parallel SPO rule symbol ==>> is used with an empty query, i.e. the left-hand side is empty. This allows us to produce a graph following the root node symbolized by ^ on the right-hand side. The default node serving as a basis of the *type graph* is *TypeRoot*. The nodes for the *type graph* are then written behind the *TypeRoot* node. As I and Bud belong to the same scale IBs (see Example 4.4.3), we enclose them in the symbols {# and #} so they form a *clique*. In our terminology for type graphs in XL, a *clique* is a set of nodes where each node is connected to all other nodes in the set with a successor edge and a branching edge. This allows successor and branching edges to be established between them. The functionality of the symbols ==>> and ::> is unaffected by multi-scale extensions.

Query and production graphs can be specified using a string of module names separated by symbols representing edges in the graph. A sequence of XL module names separated by space characters constructs a graph with nodes joined by successor edges. Square brackets indicate branching edges. For example, to query for a lateral bud (Bud) branching from an internode (I), we use the XL syntax:

```
I [Bud]
```

Extending this original syntax, if a space character separates two module names that are at different scales (the earlier coarser than the latter), a refinement edge is inserted between them. This relies on the condition that the modules have a refinement edge between them in the representing type graph. Subsequent nodes will be refined from the same node at coarser scale. A successor edge is established from the last node (from left to right) at a particular scale to the next node behind a space character at the same scale if no brackets exist between them. If the name of a module at a coarser scale follows one at a finer scale, a successor edge is connected to the coarser scale node from the last specified node, if one exists, at the same coarse scale. This syntax is illustrated in Figure 4.19. The first occurrence (from left to right)

of a node that is not enclosed in brackets at a particular scale is an element of the left-most set of nodes. In a MS production $q : L \xrightarrow{\eta,\vartheta} R$, this set of left-most nodes is the set L_L in L if the graph expressed by the statement is a query, i.e. on the left-hand side of the production. If the graph expressed by the statement is a production, i.e. on the right-hand side of the production, this set of left-most nodes is the set R_L in R.

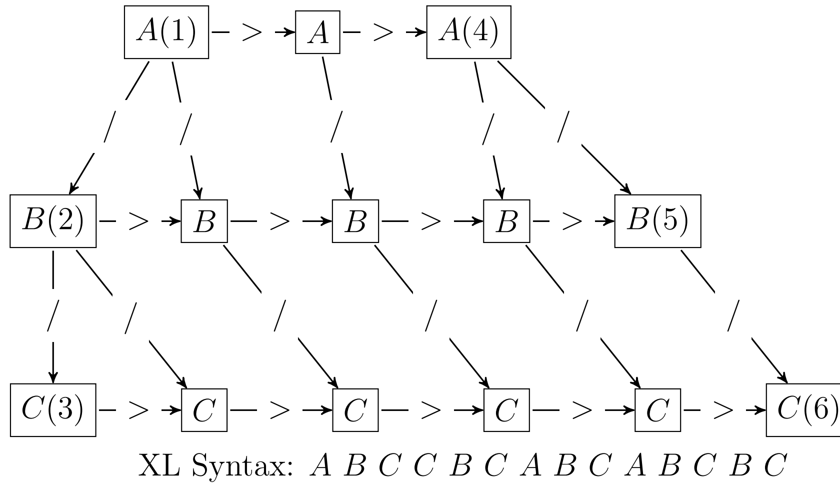


Figure 4.19: XL multi-scale syntax without branching. Module A is coarser than B, which is coarser than C. An example of a simple, acyclic query or production graph. Nodes with additional labels 1, 2 and 3 are the left-most nodes. Nodes with additional labels 4, 5 and 6 are the right-most nodes.

If an opening bracket character separates two module names that are at different scales (the earlier coarser than the latter), a refinement edge is also inserted between them. Similar to the space character, this relies on the condition that the modules have a refinement edge between them in the representing type graph. Subsequent nodes will be refined from the same node at coarser scale. A branching edge is established from the last node (from left to right) at a particular scale to the next node behind a space character at the same scale if opening brackets exist between them. If the name of a module at a coarser scale follows one at a finer scale after an opening bracket character, a branching edge is connected to the coarser scale node from the last specified node, if one exists, at the same coarse scale. For example, the production graph R_G in Example 4.4.3 illustrated in Figure 4.16 can be expressed in XL syntax as:

$$I \text{ [} Axis\ GU\ I\ Bud \text{] } GU\ I\ Bud$$

The first occurrence (from left to right) of a node that is enclosed in brackets at a particular scale with no preceeding un-enclosed node at the same scale is an element of the set of pending branch nodes. In the above statement, *Axis* and *GU* enclosed in brackets are the pending branch nodes at two distinct scales. The *I* enclosed in brackets is not a pending branch node because there is a preceeding *I* (a node at the same scale) at the beginning of the statement. In a MS production $q : L \xrightarrow{\eta,\vartheta} R$, this set of pending branch nodes is the set R_B in R if the graph expressed by the statement is the production graph, i.e. on the right-hand side of the production.

The MS production $q : L \xrightarrow{\eta,\vartheta} R$ in Example 4.4.3 is hence written in XL as shown in Listing 4.4.

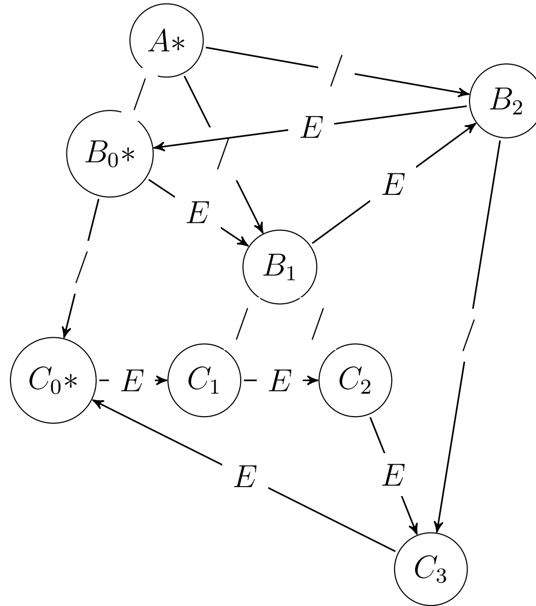
```
Bud ==> I [ Axis GU I Bud ] GU I Bud;
```

Listing 4.4: XL Multiscale Rule

Consider the query or production graph shown in Figure 4.20. A cycle is present at the intermediary scale for the nodes labelled B and at the fine scale for the nodes labelled C. The left-most and right-most nodes at the two scales with cycles can be deciphered based on the syntax of XL that brings the nodes into a strict order, e.g. b0:B_0 c0:C_0 in the first line of syntax in Figure 4.20. The left-most and right-most nodes of the graph in Figure 4.20 are the nodes labelled with an additional asterisk * because the nodes after the first comma in the code representation of the graph cannot be the left-most or right-most nodes of the query. Embedding edges can then be created based on the same connection mechanisms as in non-cyclic cases.

In some cases, refinement embedding edges must be added to embed a production graph in the remaining host graph. Consider the left-most nodes in a matching host sub-graph. Out-going refinement edges from these nodes to nodes that do not belong to the matched sub-graph are replaced with embedding refinement edges that begin from the left-most nodes of the production graph (see Figure 4.21 including syntax).

Incoming embedding refinement edges to the left-most and subsequent nodes in the production graph are created if the corresponding left-most node in the matched sub-graph had incoming refinement edges from an un-matched node (see Figure 4.22 including syntax).



XL Syntax: a:A b0:B_0 c0:C_0,
a b1:B_1 c1:C_1 -E-> c2:C_2, a b2:B_2 c3:C_3,
b0 -E-> b1 -E-> b2 -E-> b0,
c0 -E-> c1, c2 -E-> c3 -E-> c0

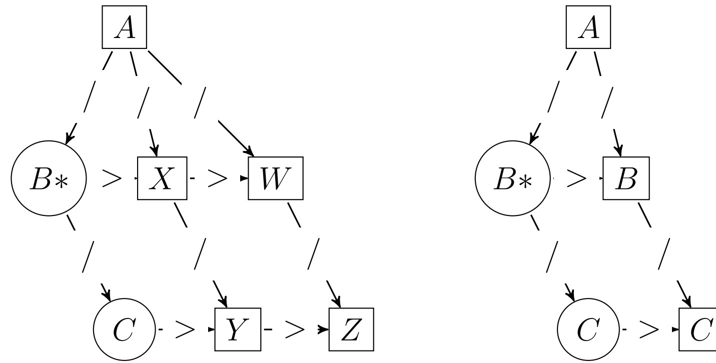
Figure 4.20: Query or production graph with cycles. Nodes labelled with an additional * are both the left and right-most nodes. Subscript numbers are not part of the syntax and only serve to help identify nodes in this figure.

Consider now the out-going refinement edges from the right-most nodes in a matched host sub-graph, connecting to nodes not in the matched sub-graph. Embedding edges are created from the right-most nodes with the same scale of the production graph to the host graph (see Figure 4.23 including syntax).

An example of the syntax for parallel rewriting is shown in Figure 4.24.

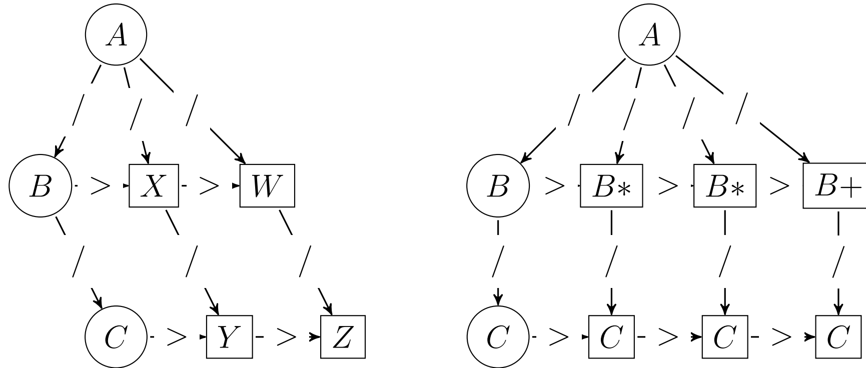
4.5.2 The Observer Programming Pattern

The combination of processes and structural developments at different scales in plant models requires additional implementation effort in comparison with single-scale models. In order to organize and reduce time taken to implement



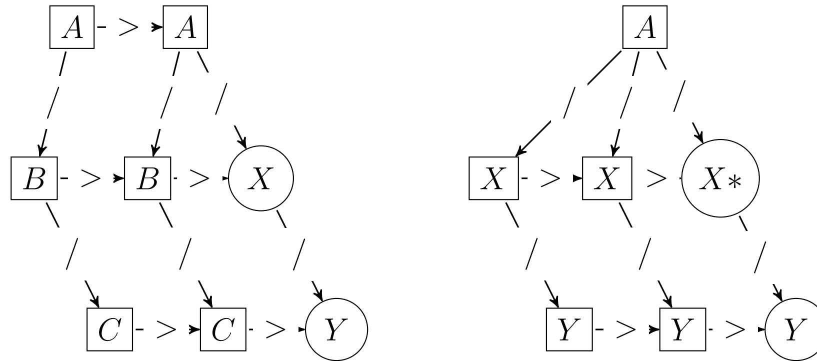
Rule: A X Y W Z ==> A B C

Figure 4.21: Syntax for rule establishing refinement embedding edges - 1. The graph on the left is transformed to the graph on the right using the specified rule. Node A is reconnected to the un-matched B* node in the original host graph. Rectangular nodes belong to the query or production graph.



Rule: X Y W Z ==> B C B C B C

Figure 4.22: Syntax for rule establishing refinement embedding edges - 2. The graph on the left is transformed to the graph on the right using the specified rule. Node A is reconnected to the left-most and subsequent B nodes labelled with * in the production graph. Node A is also reconnected to the right-most B node labelled with +. Rectangular nodes belong to the query or production graph.

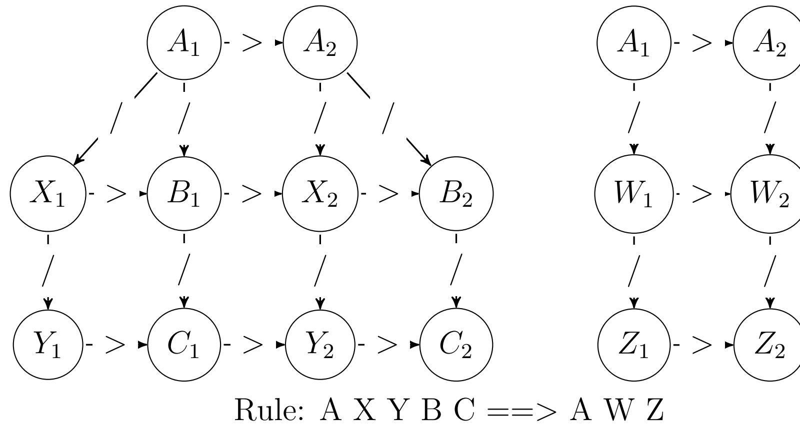


Rule: A B C A B C ==> A X Y X Y

Figure 4.23: Syntax for rule establishing refinement embedding edges - 3. The graph on the left is transformed to the graph on the right using the specified rule. Node A in the production graph is reconnected to node X labelled with an additional * in the remaining host graph. Rectangular nodes belong to the query or production graph.

these models, we propose a method [126] based on the class of formalized practices in software engineering known as design patterns [63]. In particular, our approach uses the *observer pattern*, which allows the specification of one-to-many dependencies between objects. When one object changes state, all its dependents are notified and updated automatically. In the context of the multiscale graph structures introduced in this chapter, we utilize this pattern on *scales*, i.e. elements in the structure of scales defined in Section 4.3.1. This allows the registration of dependencies between *scales* and it follows that when the state of a scale changes, all its dependent scales are notified and updated automatically.

We discuss the motivation behind the adoption of this pattern. A disadvantage of developing a model as a collection of cooperating scales is the need to integrate and maintain consistency between related scales. In object-oriented programming, such integration can be achieved by coupling the scales *tightly*, i.e. by establishing inter-scale dependencies either as member variables, indices, or method invokations. This reduces the reusability of code providing the functionality of individual scales (consider the need to extract the functionality of a scale for usage in another model). The observer pattern offers an alternative approach to establish these references without



Rule: A X Y B C ==> A W Z

Figure 4.24: Half-edge matching. The connected graph G shown on the left contains two sub-graphs matching the graph representation of $A\ X\ Y\ B\ C$. The numeric sub-scripts in the node labels are used to distinguish nodes between the two sub-graphs matched. Operators from the first sub-graph yield the set $\{A_2, X_2, Y_2\}$ while operators from the second sub-graph yield the set $\{A_1, B_1, C_1\}$. This mutual containment constitutes three 'half-edge matches'. As a result, the derived graph on the right contains edges $(A_1, >, A_2), (W_1, >, W_2)$ and $(Z_1, >, Z_2)$.

tight coupling. It uses two distinct roles: *subject* and *observer*. A subject can have multiple dependent observers. The observers are notified whenever the subject changes its state. This kind of interaction is also known as *publish-subscribe*. The subject is the publisher of notifications. It sends notifications without knowledge of the identity of the observers while the observers subscribe to notifications.

A precondition for the usage of the observer pattern is the possibility to represent a structure of scales in the XL program. We allow the declaration of a *scale* in XL code as:

```
scale A;
scale B;
scale C;
```

where `A`, `B`, and `C` are scale labels. The syntax is identical to the declaration of *modules* in XL but does not support further extension from superclasses.

Once the required scales are declared, the structure of scales can be constructed as a subgraph connected to the root of the main graph in the `init()` method as shown in Listing 4.5.

```
int ScaleToType=1;

scale Tree;
scale Organ;

module T;
module I;

protected void init()
[
  Axiom ==> T /> I;
  ==>> ^ [/> SRoot /> tr:Tree /> or:Organ]
          /> TypeRoot /> t:T /> i:I
         tr-ScaleToType->t,
         or-ScaleToType->i
     ;
]
```

Listing 4.5: Structure of scales construction

In Listing 4.5, a structure of scales with coarse to fine linear ordering of scales `Tree` to `Organ` is constructed and connected to `SRoot`, the root node of the structure of scales with refinement edges. Two corresponding types, i.e. *modules* in terms of XL code, are used to construct the type graph connected to `TypeRoot` with refinement edges. To relate the scales with their respective types, an arbitrary edge label (not a successor, branching, or refinement edge) can be used. In Listing 4.5, the edge label `ScaleToType` is used to connect the nodes in the structure of scales to the type graph nodes. The code in Listing 4.5 also initializes the instanced graph with two nodes, one from each type.

The syntax to establish dependencies between scales is:

```
x.observe(y,code,"m");
```

where `x` is the observer, `y` is the subject, `code` is an integer value identifying a notification message, and `m` is the member method of the observer `x`

to invoke when `y` publishes a `code` notification.

We now extend the code in Listing 4.5 to establish a dependency of the `Tree` scale on the `Organ` scale. The extended code is shown in Listing 4.6.

```
int ScaleToType=1;

int EVOLVE=0;       //notify code 0
int EXTRAPOLATE=1; //notify code 1

scale Tree
{
  public void grow(){notify(EXTRAPOLATE);}
};
scale Organ;

module T;
module I;

protected void init()
[
  Axiom ==> T /> I;
  ==>> ^ [/> SRoot /> tr:Tree /> or:Organ]
          /> TypeRoot /> t:T /> i:I,
         tr-ScaleToType->t,
         or-ScaleToType->i,
      {
        this.observe(or, EVOLVE, "internodeGrow");
        this.observe(tr, EXTRAPOLATE, "treeGrow");
        tr.observe(this, EXTRAPOLATE, "grow");
      }
     ;
]

public void run()[
  o:Organ ::> {o.notify(EVOLVE);}
]

public void internodeGrow()[
```

```
  //.. rules for organ scale (internode) growth
  {this.notify(EXTRAPOLATE);}
]

public void treeGrow()[
  //.. rules for tree scale growth
]
```

Listing 4.6: Establishment of observer pattern in XL

The differences between Listing 4.5 and Listing 4.6 are highlighted in this paragraph to illustrate how dependencies between the scales `Tree` and `Organ` are established in XL. Firstly, two notification identifiers, `EVOLVE` and `EXTRAPOLATE` are declared as integers. A method `grow()` in the scale declaration of `Tree` is defined. This method sends out a notification with code `EXTRAPOLATE` when invoked. In the `init()` method, the rule is appended with three lines of invokation to the `observe` method. The first two lines establish observations by the RGG class (the code in the Listing is actually an extension of a class "RGG") on the scales `Organ` and `Tree`. This allows the invokation of the methods `internodeGrow` and `treeGrow` based on notifications from the scales. The third line establishes an observation by the `Tree` scale on the RGG class. This allows the invokation of the method `grow` based on notifications from the RGG class. The flow of the dependencies at runtime is as follows:

- `run()` is invoked. The scale `Organ` bound to variable `o` is queried. The organ scale publishes a notification by invoking the method `o.notify(EVOLVE);` on the node `o`.
- The RGG class observes the notification with code `EVOLVE` from the organ scale. As a result, the method `internodeGrow` is invoked. Some rules for organ scale development are run. Next, the RGG class publishes a notification with code `EXTRAPOLATE`.
- The tree scale (`Tree` node in structure of scales) observes the notification with code `EXTRAPOLATE`. As a result, the method `grow` is invoked. In the `grow` method, the `Tree` scale publishes a notification with code `EXTRAPOLATE`.
- The RGG class observes the notification with code `EXTRAPOLATE` from the tree scale. As a result, the method `treeGrow` is invoked. Rules

for tree scale development (likely based on developments at the organ scvale) are run.

In this manner, the methods `internodeGrow` and `treeGrow` are not aware of each other's existence and they can be copied and reused in another model easily. In addition, only one line of notification is necessary for the invokation of methods of multiple observers. In contrast, consider the case with multiple observers where no observer pattern is used. Multiple method invokations must occur in the subject's method, possibly one for each observer. Most importantly, such a design pattern encourages the segregation of functionality based on scales for individual methods. We observe such segregation in Listing 4.6 where `treeGrow` and `internodeGrow` provide functionality at tree scale and organ scale respectively without overlaps. This syntax implementation for the observer pattern in XL is a preliminary solution that should be improved to become concise and transparent to the modeller or programmer.

4.6 Technical Documentation

In this section, the software implementation of the techniques in this chapter within the software GroIMP [91] is documented. They are implemented as extensions to the existing open source code on:
http://sourceforge.net/projects/groimp/. For convenience, we shall call them *multiscale extensions*. All lines of code removed or added have been marked at the start by the comment *//multiscale begin* and at the end by the comment *//multiscale end*. All classes added have been commented with the remark *This class is part of the extension of XL for multiscale modelling*.

In the following sub-sections, an overview of the use cases is first given, followed by descriptions of classes grouped by functionality. The purpose of this section is to provide the reader with an idea of what key classes contribute to the multiscale extensions. Detailed comments can be found in the source code itself.

4.6.1 Use Cases

Figure 4.25 shows the use case diagram for a user (the actor) of GroIMP. While the features and functionalities of GroIMP are manifold, the diagram focuses on the aspects changed by the extensions for multiscale rewriting

and modelling. The rest of the functionalities are encompassed within the "Use other functionalities of GroIMP" use case. The top use case, "Write /

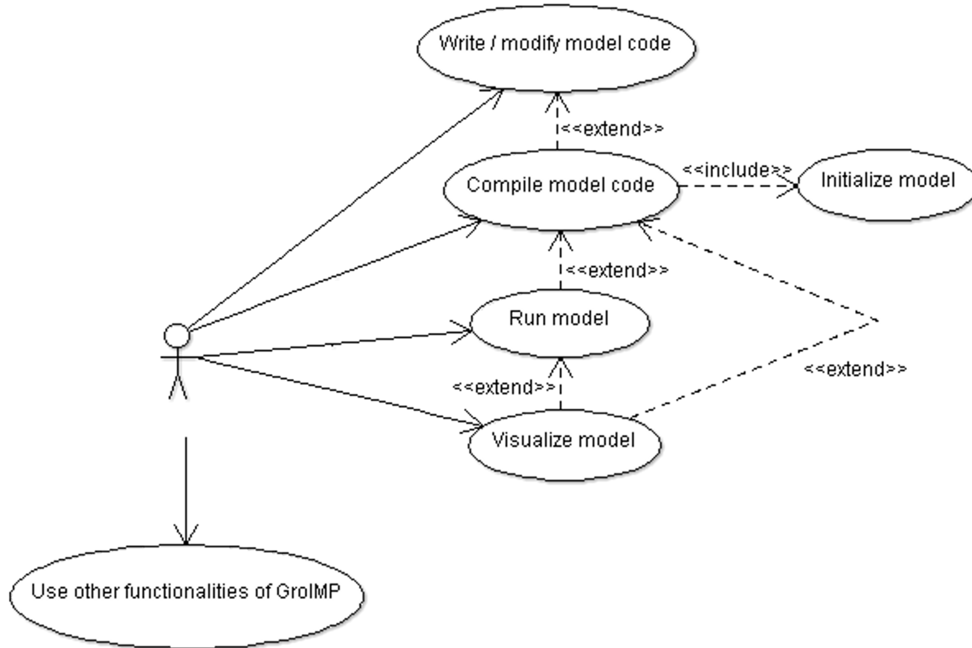


Figure 4.25: Use case diagram for user of GroIMP.

modify model code" refers to the writing of XL code in GroIMP. This use case now includes specifying a structure-of-scales, a type graph, and writing multiscale rules as described in Section 4.5.

The second use case "Compile model code" refers to the compilation of written code. Compilation is triggered whenever the user presses "save" on the graphical user interface. By default, GroIMP runs the method *init()* immediately after compilation. Hence the use case "Initialize model" is included in the "Compile model code" use case.

The third use case "Run model" refers to the execution of a certain method in the written code. This is triggered whenever the user presses on a button named after a public method on GroIMP's graphical user interface. As a result, the model runs and produces results in the form of data in the running program.

The fourth use case "Visualize model" refers to the 3D visualization of the model. This is usually within a 3D panel in GroIMP that is refreshed

with every invokation of a method in the model code.

4.6.2 Compilation

Several classes have been modified and added for the "Compile model code" use case. We illustrate them in Figure 4.26. The majority of the classes modified and added in the multiscale extensions for compilation lie in the plugins "RGG" and "XL-Compiler". The classes involved in tokenizing and parsing code to generate abstract syntax trees (ASTs) are found in the package "de.grogra.xl.parser" (seen at the top of Figure 4.26). The classes involved in traversing abtract syntax trees (ASTs) to create expression trees using a three pass approach (see chapter 8.3.1 in [91]) are found in the package "de.grogra.xl.compiler", which is seen in the middle of Figure 4.26. The classes that trigger compilation are found in the package "de.grogra.rgg.model". They have not been modified but are shown in Figure 4.26 for reference purposes.

GroIMP uses the ANTLR (www.antlr.org) tool for tokenizing code and parsing code. Notice that the class "Parser" extends from the class "antlr.LLKParser" from the ANTLR library. The classes "JavaTokenizer", "XLParser", "XLTokenizer" and "XLTokenTypes" are generated automatically whenever changes to the grammar file "XL.g" are made and when the ant build script (build.xml) for the "XL-compiler" plugin is run. For multiscale extensions, the keyword *scale* and the symbols for enclosing nodes in a clique, {# and #} have been added to "XL.g". In addition, an ANTLR rule "scaleDeclaration[AST mods]" for declaration of scales similar to the declaration of modules is added. A list of edge or node tokens in a clique can be written to the AST sub-tree containing a production statement (see line starting with "productionStatement[AST prev]" in "XL.g").

For writing expression trees, GroIMP uses the ANTLR tool. Notice that the class "CompilerBase" extends from the class "antlr.TreeParser" from the ANTLR library. The classes "Compiler", "CompilerBase" and "CompilerTokenTypes" are generated automatically whenever changes to the grammar file "Compiler.tree.g" are made and when the ant build script (build.xml) for the "XL-compiler" plugin is run. For multiscale extensions, the ANTLR rules "compilationUnit[...]" and "classDecl[...]" in "Compiler.tree.g" are modified so a *scale* declaration is a class declaration in Java, just like how a *module* declaration is a class declaration. In addition, the ANTLR rule "stat[...]" is modified so that methods in the "Producer" class (see Section 4.6.3) are

invoked when the symbols representing a clique are encountered. This allows the program to establish necessary edges between nodes in a clique during production.

XL syntax for query graphs are written as sequences of *patterns* in expression trees during compilation. For details on *patterns* and *places*, see Section 17.3.1 in www.grogra.de/xlspec and [91]. A query is composed of patterns like node patterns or edge patterns, together forming compound patterns. During the three-pass expression tree writing stage of compilation, the "PatternBuilder" class is responsible for creating these patterns. To allow for the multiscale syntax of spacings between nodes at different scales, a new class "SpacingPattern" is added, which extends the original "EdgePattern". More specifically, the methods "addPattern", "add", and "join" in the "PatternBuilder" class are invoked during expression tree generation to create instances of the "SpacingPattern" class.

4.6.3 Run

The main packages involved in queries and productions at runtime are "de.grogra.xl.query", "de.grogra.xl.impl.base", "de.grogra.xl.vmx", and "de.grogra.rgg". These packages are in the plugins "XL", "XL-Impl", "XL-VMX", and "RGG" respectively. Figure 4.27 shows the classes modified and added in the multiscale extensions. We describe now the flow of events programmatically when a query in XL is evaluated at runtime. The method "findMatches" in the class "Query" (found in the package "de.grogra.xl.query" in the middle of Figure 4.27) is invoked. This method begins to construct "Matcher" objects corresponding to individual patterns by invoking "getMatcher" in the "Query" class and subsequently the overriden "createMatcher" methods of individual patterns (to avoid cluttering, Figure 4.27 shows only the "SpacingPattern". There are however other patterns referenced from "CompoundPattern" such as "NodePattern", "EdgePattern", etc.). Consequently, these "Matcher" objects are gathered as a linked-list headed by a "CompoundPattern.Matcher" object. This linked-list is found in the "QueryState" class. For multiscale extensions, the "createMatcher" method is implemented in the "SpacingPattern" class and it supports the creation of both single and multiscale matchers depending on the prescence of a type graph. Once the linked-list of "Matcher" objects is generated, the method "findMatches" in the "QueryState" class is invoked and it starts to iterate through the linked-list of "Matcher" objects. While iterating through

the linked-list, the "findMatches" method of each "Matcher" object is invoked, binding graph nodes (and edges) to variables in a simulated call-stack (Kniemeyer implemented a simulated method call stack under the package "de.grogra.xl.vmx" because the low level Java call stack is inaccessible at program level. See Kniemeyer's thesis [91] for details). These variables are instances of the "VMXState.Local" class. If all the "Matcher" objects in the linked-list managed to bind their respective patterns to call stack variables, the "Producer" object is called to start production.

In the multiscale extensions, the "QueryState" class is extended with a reference to a "QueryStateMultiScale" class that contains information for multiscale queries. The information consists, for example, of lists of node-edge-node tuples, where the node pairs are at the same scale or different scales. These tuples correspond to the successively bound spacing or edge patterns described in the previous paragraph.

The methods "producer$beginExecution" and "producer$endExecution" of the "Producer" class in the "de.grogra.xl.impl.base" package are invoked at the start and at the end of rule productions. The extensions for multi-scale rule productions have also been implemented in these methods of the "Producer" class. Several member variables for containing references to the left-most and right-most nodes at each scale have been added. In addition, nodes belonging to a clique are also traced within lists in the class. The "Producer" class operates using a collection of queues, each representing a command such as add node, delete node, add edge, delete edge. For each rule, the nodes and edges of the query and production graphs, as well as embedding edges are added to the queues. The queues are executed at the end when all possible query matches have been found. The multiscale extensions build upon this previous mechanism for multiscale embedding, filling up the queues with necessary commands based on the theory described earlier in this chapter. The extensions are however backward compatible, i.e. old methods are used if there is no type graph in the graph data structure.

A type graph is specified in the data structure as a rooted graph with root node of the class "TypeRoot" from the package "de.grogra.rgg". A structure of scales is specified in the data structure as a root graph with root node of the class "SRoot" from the package "de.grogra.rgg". These root nodes are expected to be child nodes of the original graph root node. For relating nodes in the type graph, i.e. types, to the respective scales in the structure of scales, a custom edge type must be used to connect a scale node (instance of "Scale" class in the "de.grogra.graph.impl" package) in the structure of

scales to a node in the type graph. The partial ordering of the structure of scales is established using the unique refinement edge type.

To accelerate queries to the type graph, for example to check if a node is of a finer scale than another, a series of cache structures are constructed within the "Graph" class in the "de.grogra.xl.impl.base" package. Without these cache structures, the type graph must be traversed every time a query to the type graph is made. The following cache structures have been implemented:

- HashMap<Type, HashMap<Type, Boolean> > cacheScaleSame;
- HashMap<Type, HashMap<Type, Boolean> > cacheScaleComparable;
- HashMap<Type, Object> cacheTypeNode;
- HashMap<Type, HashMap<Type, Integer> > cacheMinEncoarseDiff;

The structures are used to trace if two types are at the same scale, at comparable scales, and if comparable, the minimum number of scale encoarsements between them. The Hashmap "cacheTypeNode" is used to retrieve the node in the type graph for a specific type.

4.6.4 Visualization and the Observer Pattern

GroIMP interprets the main graph data structure as a scene graph for 3D visualization. For rendering purposes, it utilizes the visitor pattern for scene graph traversal. The main class for rendering logic is hence the "DisplayVisitor" class in the package "de.grogra.imp3d". This class plays the role of a visitor and for multiscale extensions, it works together with several other classes shown in Figure 4.28. For every graph node traversed, the classes "View3D" and "ViewConfig3D" are consulted to see if the node is visible by invoking the method "isInVisibleScale". This method retrieves the root of the main graph data structure and the corresponding "Scale" node (of class "de.grogra.graph.impl.Scale") in the structure of scales. For each "Scale" class node or object, there exists a flag to indicate if the scale is visible or not. These flags can also be set via the graphical user interface implemented via the class "ScaleVisibilityPanel" in the "de.grogra.imp" package. Only visible nodes during scene graph traversal are rendered.

The observer pattern is implemented using Java's "Observer" interface in the "java.util" package. Classes implementing this interface can observe one

another by invoking the "addObserver" method of the object being observed and passing in the observer object as argument. Upon the establishment of such observation relationships, observed objects can invoke the method "notify", which results in the invokation of the method "update" in all observers of the observed object. Because we want to allow scale-scale observations, each node or object of the "Scale" class in the "de.grogra.graph.impl" package consists of a "ScaleObserver" variable that implements Java's "Observer" interface. To establish scale-scale observation, we invoke the "observe" method in the "Scale" class and pass in the observed object, the notification key in the form of an integer, and the method to invoke when the observed notifies with the key. The "ScaleObserver" class contains reference to the "Scale" class object or node and a hash map containing the registered methods to invoke for particular observed objects and notification keys whenever the "update" method is called.

Remark: The contents in Sections 4.2, 4.3, 4.5.1, and 4.1 are published in [128], [125], and [130].

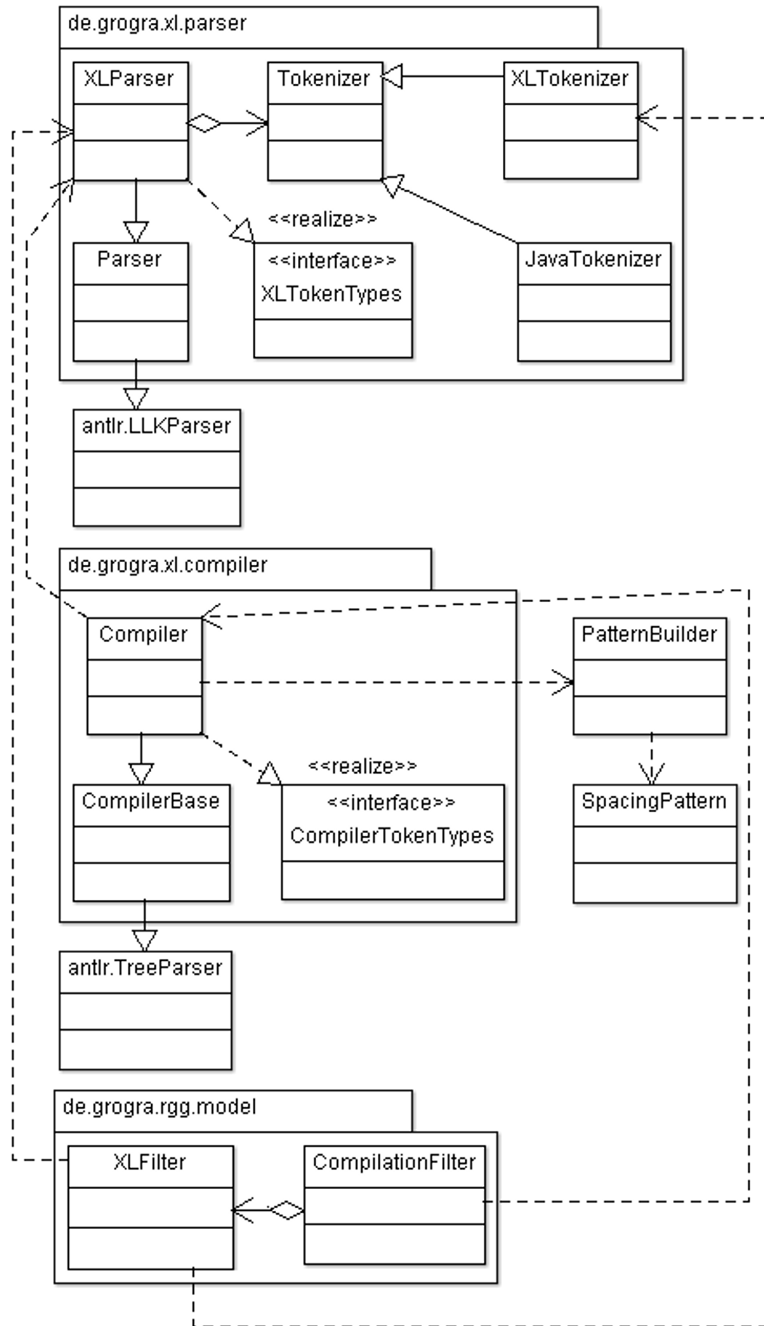


Figure 4.26: Class diagram for classes used in compilation.

Figure 4.27: Class diagram for classes modified and added for running multiscale XL code.

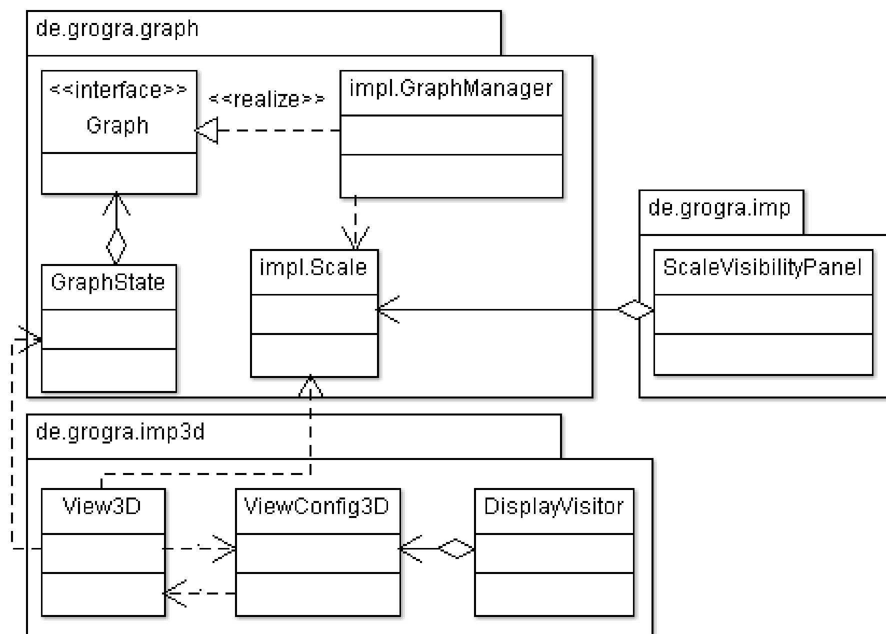


Figure 4.28: Class diagram for multiscale visualization.

Part II

Level-of-Detail (LOD) Visualization

Chapter 5

Multiscale & LOD Visualization

5.1 Incremental LOD for Branching Structures

The development of branching structures occurs in many aspects of nature [69]. This section presents an incremental computation method for the level-of-detail (LOD) visualization of dynamic branching structures. Our method is the first to minimize LOD computation delays occuring between simulation steps of branching structures that grow. As a result, transition between steps is less choppy, giving the viewer a smoother visual feedback of the development processes. Our method applies to grammar or L-system based specifications of branching structures, and is therefore particularly useful for plants or vegetation scenes.

Our LOD computation method is based on the notion of incremental computation [5, 23] and consists of buffers for polygonal areas between simplified and non-simplified polylines or branch ramifications. Each buffer stores the quadratic error for the simplification of a polyline or a pair of lines at a ramification point, thus capturing detail for a given error threshold. Transitioning between levels-of-detail is achieved by selective traversal of a multiscale instanced graph (Section4.3.3) that arranges the buffers in a hierarchical manner. Our method computes quadratic error incrementally by replacing and appending to the last term of the shoelace formula (also known as Gauss's area formula or the surveyor's formula) [15] in order to avoid re-visiting all

coordinates as the structure develops.

By coupling the incremental LOD computation with a multiscale scene graph representation, a dynamic multi-resolution branching structure can be rendered at high frame rates.

This section presents an LOD computation method and a multi-resolution model for branching structures. In order to achieve these objectives, we introduce the following techniques:

- An incremental computation method for estimating quadratic errors resulting from polyline simplification.
- A quadratic error estimate for filtering branch ramifications. As ramifications below an error threshold are filtered away, the branching structure is "pruned" and only decipherable ramifications are rendered.
- A multiscale graph representation of branching structures. As opposed to conventional scene graphs, this data structure is utilized in two aspects: local and global. These aspects segregate the topology of individual branching structures defined by rules (local) from the scene construction (global).

The approach in this section has the following drawbacks and limitations:

- Accuracy of the error estimates is subjected to the modeller's decision on the number of scales (hierarchical levels) and ratio between objects at different scales in the data structure.
- Accuracy of the error estimates does not match up to some existing LOD or simplification algorithms.
- The thickness of lines in the branching structures is not accounted for.

5.1.1 Previous Work

Many methods for computing LODs of branching structures have been proposed in the past. The earlier methods are generic simplifications, mostly applicable to vertex-edge constructs of geometry. More recent researches explore simplifications on alternative representations, such as topology (connectivity between segments in branching structures), textures, volumetric representations and hybrid representations.

Early techniques relevant to branching structures are line simplification algorithms. McMaster categorized and compared some of these methods [117], which were revisited with newer algorithms by Shi and Cheung in a more recent review [160]. The methods are categorized based on the range of neighbouring points considered in the algorithm. In an order of ascending range, some algorithms include the nth point routine, the routine of distance between points, the perpendicular distance routine, the Reumann-Witkam routine [146], the Zhao-Saalfeld algorithm [179], the Opheim simplification algorithm [131], the Lang simplification algorithm [98], the Douglas-Peucker simplification algorithm [44] and the Visvalingam-Whyatt algorithm [174]. While these methods are generally good in applications with requirements for high fidelity, none of them allow filtering or substitution of deeper parts (in terms of branching depth) in the branching structure.

Alternatively, research in the rendering of plants also saw advancements in the simplification of branching structures. Weber and Penn [176] proposed the substitution and filtering of finer twigs with increasing viewing distance. Marshall et al. [109] described another technique with discrimination against higher branching orders or depths. Following the dominance of polygonal geometry and the initial idea of replacing primitives with quadric surfaces by Gardner [64], Max [116] and Meyer and Neyret [119] used z-buffers and volumetric textures respectively to represent botanical objects. However, problems like parallax error and loss of detail at close viewing distances were generally identified with texture-based and billboard approaches. Polygonal simplifications, on the other hand, were posing limitations to frame rates for large scenes. As a result, many improvements and extensions are made. For example, the geometry substitution and filtering approach was extended by Deussen et al. [43] for plant populations. Meyer et al. [120] proposed to use a hierarchy of bi-directional textures and Decaudin and Neyret [40] extended the usage of volumetric textures to forest scenes.

Some recent methods focus on filtering the branching structure using more specific criterias. For example, Clasen and Prohaska [31] used branching angle and branch size to successively merge lines in the skeleton of the branching structure. Lluch et al. [104] sorted the branching structure according to lengths from root to leaf. Both approaches construct a hierarchy according to criterias in order to filter away parts of the branching structure for lower resolutions. While these methods have clear criterias, a direct relationship between view-dependent pixel error and simplification error is not present. In addition, given a growing branching structure, the criteria-based

hierarchies require repeated reconstructions.

Other recent methods can be perceived as hybrid approaches with breakthroughs in terms of efficiency and realism. To complement a high-speed hierarchical simplification of foliage, Deng et al. [42] proposed the usage of polyline simplifications on branching structure skeletons before multi-resolution mesh generation. Livny et al. [103] introduced lobe textures suitable for representing higher order branches while keeping lower order branches as a skeletal graph. Also using a combination of approaches, Fan et al. [53] used a hierarchy based on branching order to filter away high order branches for low resolution models of trees while retaining fine twigs as textures together with foliage. However, to the best of our knowledge, none of the methods address dynamic structures and most require extensive pre-computations. Moreover, not all techniques are catered for grammar or L-system generated structures, some requiring intermediary conversions.

The cache in our incremental approach updates on-the-fly and avoids complete reconstructions. The incremental error estimation is based on existing polyline simplifications, such as the perpendicular distance routine and the Douglas-Peucker simplification algorithm. These algorithms use the distance between points on the simplified and original line as simplification error. We continue the line of research from Clasen and Prohaska and Lluch et al. towards criteria-based filtering of multi-resolution branching structures. By relating pixel error and estimated errors at branch ramification points, line segments are filtered for low resolution representations. Our methods do not use textures in order to avoid costly alignment and scaling of textures to a dynamic structure.

The multi-scale graph data structure in Chapter 4 is used to represent the scene with branching structures, supporting graph grammar usage. We interpret the graph in two ways: a conventional scene graph and multiscale branching topologies. Although this data structure was designed under the motivation of multiscale plant representation [67], we explicitly use the term "branching structures" to include non-botanical objects.

5.1.2 Polyline Incremental LOD and Ramification LOD

In this section, we first describe a method to compute estimated error for polyline simplifications. As a polyline is extended with more line segments at the end, the computation proceeds incrementally from previous estimations. Secondly, we present an error estimation method for ramifications.

5.1.2.1 Polyline Incremental LOD

A 2-dimensional (2D) polyline P is a sequence of n vertices $(v_1, v_2, ..., v_n)$, where v_i is a vertex with 2D cartesian coordinates (x_i, y_i), $1 \leq i \leq n$, so that P consists of line segments connecting the consecutive vertices. The straight line Q from the first vertex v_1 to the last vertex v_n is a simplified representation of the polyline P with quadratic simplification error A, where A is the signed polygon area enclosed by P and Q (Figure 5.1). Specifically, based on the surveyor's formula [15],

$$A = \frac{\sum_{i=1}^{n-1}(x_i y_{i+1} - y_i x_{i+1}) + (x_n y_1 - y_n x_1)}{2} \quad (5.1)$$

where $(x_i, y_i) = v_i$, $v_i \in P$, and $n \geq 3$. Let P_t be a polyline at time step t,

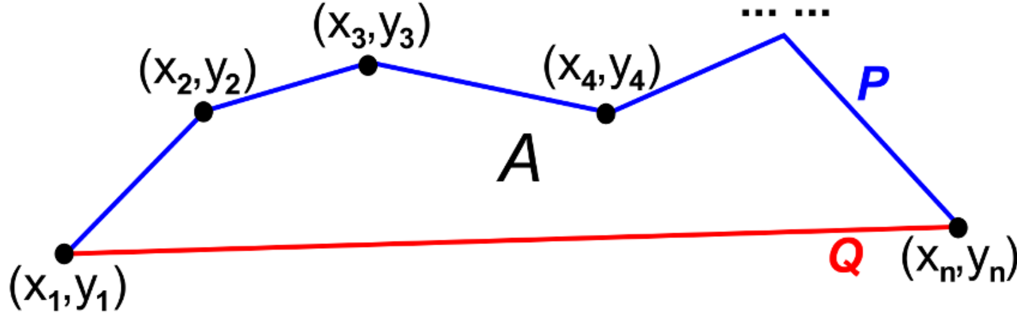


Figure 5.1: Polyline P, simplification straight line Q, and enclosed polygon area A used as simplification error.

P_{t+1} be the polyline at time step $t+1$ and $|P|$ be the notation for the number of vertices in an arbitary P. The sequence of polylines $(P_1, P_2, ..., P_k)$ with k time steps then describes a "growing" or dynamic polyline. The dynamic polyline is such that $|P_t| < |P_{t+1}|$, and $\forall i \in \{1, ..., |P_t|\}$: $a_i = b_i$, where $a_i \in P_t$, $b_i \in P_{t+1}$ and $1 \leq t \leq k-1$.

The simplified lines Q_t and Q_{t+1} for P_t and P_{t+1} yield quadratic simplification errors A_t and A_{t+1} respectively (Figure 5.2). As A_t is computed prior to the computation of A_{t+1}, we can compute the quadratic simplification error at time step $t+1$ incrementally by

$$A_{t+1} = A_t + \Delta A_{t+1} \quad (5.2)$$

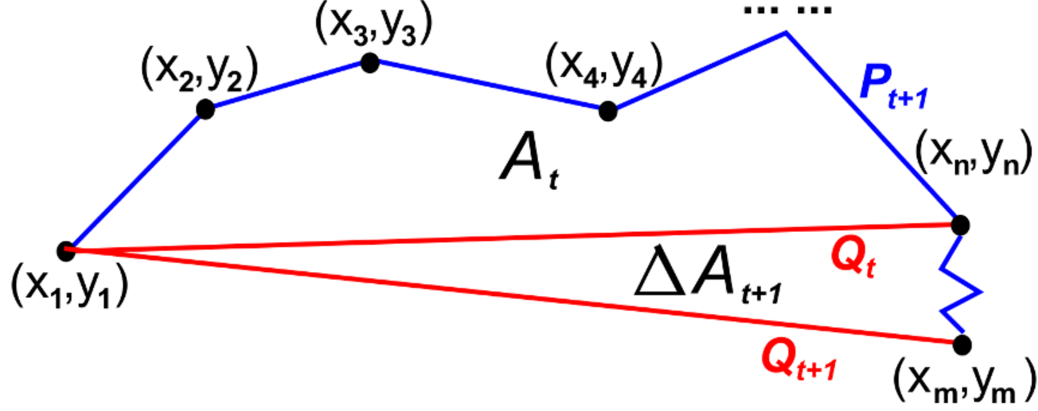


Figure 5.2: Polyline P_{t+1} is shown in blue. The simplification lines Q_t and Q_{t+1} for time steps t and $t+1$ respectively are shown in red. The *incremental term* ΔA_{t+1} is the polygon area enclosed by (x_1, y_1), (x_n, y_n), and (x_m, y_m). Simplification error for time step $t+1$ is the polygon area computed by the sum of A_t and ΔA_{t+1}.

where ΔA_{t+1} is the *incremental term*. Let $|P_t| = n$ and $|P_{t+1}| = m$. The *incremental term*, specifically, is

$$\begin{aligned}\Delta A_{t+1} =& A_{t+1} - A_t \\ =& \frac{1}{2}\left[\sum_{j=1}^{m-1}(x_j y_{j+1} - y_j x_{j+1}) + (x_m y_1 - y_m x_1)\right] - \\ & \frac{1}{2}\left[\sum_{i=1}^{n-1}(x_i y_{i+1} - y_i x_{i+1}) + (x_n y_1 - y_n x_1)\right] \\ =& \frac{\sum_{j=n}^{m-1}(x_j y_{j+1} - y_j x_{j+1}) + (x_m y_1 - y_m x_1)}{2} - \\ & \frac{(x_n y_1 - y_n x_1)}{2}\end{aligned} \tag{5.3}$$

In general, the quadratic simplification error A_q of Q_q for P_q at an arbitrary time step q in incremental terms is

$$A_q = \begin{cases} A_1 + \sum_{i=2}^{q} \Delta A_i & \text{if } q \geq 2 \\ A_1 & \text{if } q = 1. \end{cases} \tag{5.4}$$

The above methods for 2D polylines are extended for 3-dimensional (3D) polylines. A 3D polyline P is a sequence of n vertices $(v_1, v_2, ..., v_n)$, where v_i is a vertex with 3D cartesian coordinates (x_i, y_i, z_i), $1 \leq i \leq n$, so that P consists of line segments connecting the consecutive vertices. As in the case of 2D polylines, the straight line Q from the first vertex v_1 to the last vertex v_n is a simplified representation of P. The quadratic simplification error of Q in the xy-dimensions (*xy-dim*) is

$$A_{xy\text{-}dim} = \frac{1}{2}\left[\sum_{i=1}^{n-1}(x_i y_{i+1} - y_i x_{i+1}) + (x_n y_1 - y_n x_1)\right], \tag{5.5}$$

and analogously for the yz-dimensions (*yz-dim*) and xz-dimensions (*xz-dim*). Similar to 2D polylines, they can each be expressed incrementally using equation (5.2). The consolidated quadratic simplification error for Q is

$$A = max(A_{xy\text{-}dim}, A_{yz\text{-}dim}, A_{xz\text{-}dim}) \tag{5.6}$$

5.1.3 Ramification LOD

Let a parent line in 3D be specified by a starting vertex $v_s = (x_s, y_s, z_s)$ and an ending vertex $v_e = (x_e, y_e, z_e)$. A *ramification* is the protrusion of a branching line at a vertex v_r along the parent line, so that $v_r = (x_s + (x_e - x_s)t, y_s + (y_e - y_s)t, z_s + (z_e - z_s)t)$ and $0 \leq t \leq 1$. The branching line has starting vertex v_r and an ending vertex v_b that does not lie along the parent line. Branching lines can be parent lines for further ramifications, resulting in a branching structure. We discuss the representation of an extensive branching structure in section 5.2.

The quadratic error estimate A for a *ramification* is the area of the triangle with vertices v_s, v_r, and v_b. Specifically,

$$A = \frac{1}{2}|v_r v_s||v_r v_b| \sin\theta, \tag{5.7}$$

where $|v_r v_s|$ is the distance from v_s to v_r, $|v_r v_b|$ is the length of the branching line and θ is the angle as shown in Figure 5.3.

5.2 The Multiscale Graph and Grammar

In this section, we first describe the generation and modification of a multiscale graph data structure that represents both a scene as well as branching

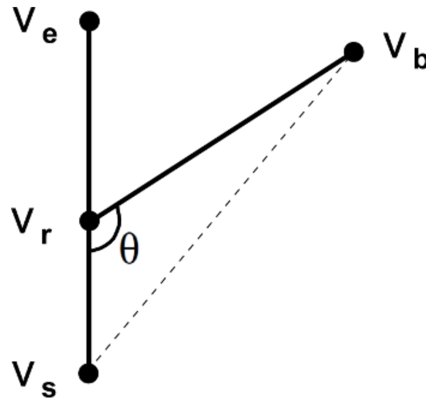


Figure 5.3: Parent line from v_s to v_e with ramification of branching line from v_r to v_b. The area of the triangle with vertices v_s, v_r, and v_b is used as error estimate of the ramification.

structures within the scene. The graph is managed in a local aspect concerned with individual branching structures, and a global aspect concerned with scene organization and hierarchies. We present these aspects separately, each with key graph grammar rule statements in the programming language XL [96, 128]. Secondly, we present the traversals of the graph for rendering, including the update and extraction of resolution specific geometry based on the error estimations, in Subsection 5.2.3.

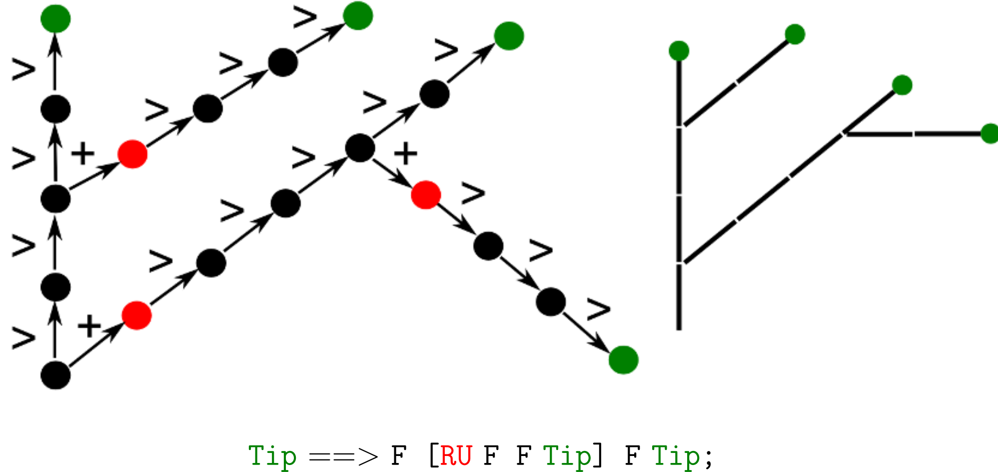
5.2.1 Local - The Multiscale Branching Structure

Unsimplified branching structure: Before describing an extensive hierarchy of simplifications for a branching structure, we first illustrate the data structure for an unsimplified branching structure based on L-systems [102, 145] and turtle geometry [4]. The turtle commands F and RU represent a forward movement and a rotation around an axis orthogonal to movement respectively. With an additional graph node type *Tip*, which represents an ending tip of the branching structure, the following rule statement in XL generates a graph data structure for an unsimplified branching structure procedurally:

```
Tip ==> F [RU F F Tip] F Tip;
```

Figure 5.4 shows the geometric interpretation against the graph data structure after two applications (time steps) of the above rewriting rule. Rami-

fications indicated by square brackets establish branching edge connections labelled +, while successions indicated by an empty space establish successor edge connections labelled >.



```
Tip ==> F [RU F F Tip] F Tip;
```

Figure 5.4: Results after applying rule twice. Left: Graph data structure. Green nodes are *Tip* nodes, black nodes are *F* nodes, and red nodes are *RU* nodes. Right: Geometric interpretation of the graph.

Polyline simplifications: A sequence of nodes connected consecutively by successor edges can be interpreted as a polyline. To introduce simplifications to polylines represented in the graph, multiscale rule statements are used in XL. For example, a module named *Sim* denoting the first level of polyline simplifications can be introduced. A type graph is required in XL for the interpretation of multiscale rule statements. In this case, we construct a type graph before invoking other procedures. It is constructed with the statement:

```
^ /> TypeRoot /> Sim /> {# F RU Tip #};
```

where ˆ is the root node of the graph data structure, *TypeRoot* is the root node of the type graph, and the symbols /> are directed *refinement edges* that connect coarse scale nodes to fine scale nodes. Since *Sim* nodes are simplifications of the finer polylines, they are coarse scale or coarse resolution representations of subgraphs composed of the nodes *F*, *RU*, and *Tip*. The

symbols **{#** and **#}** enclose nodes in a clique, establishing successor and branching edges bi-directionally between every pair of nodes enclosed. Edge connections in the type graph dictate valid edge connections in the graph data structure. Figure 5.5 illustrates the type graph resulting from the above statement. Given a type graph, multiscale rules can specify branching struc-

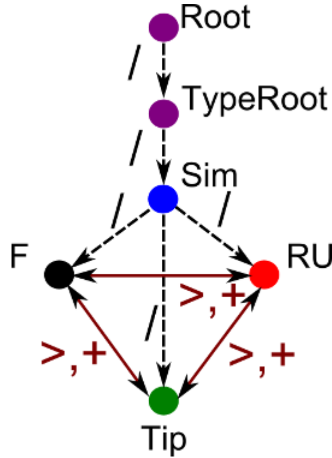


Figure 5.5: Illustration of type graph. Black, dotted edges labelled / are refinement edges. Colored, solid edges are successor (>) and branching (+) edges that connect nodes in a clique.

tures with polyline simplifications. For example, we append the earlier rule statement with *Sim* nodes to simplify every two *F* nodes and occasionally an *RU* node in the unsimplified structure to a *Sim* node:

```
Tip ==> Sim F [Sim RU F F Tip] F Tip;
```

The two *F* nodes not enclosed in brackets form a simple polyline with two line segments. This polyline is simplified to the left-most *Sim* node. The *RU* node and the two *F* nodes between the brackets are simplified to the *Sim* node between the brackets. These simplifications are represented by refinement edges in the graph data structure. In addition, successor and branching edges are established to embed newly created nodes in the graph (following embedding mechanisms in Chapter 4). Figure 5.6 shows the development of the branching structure in this example for two time steps. The subgraph consisting of all *Sim* nodes and the edge connections between them represent one level of simplification for the unsimplified branching structure.

More than one level of simplification can be introduced in the same manner.

Ramification axes and the encompassing node: Polyline simplifications can be applied to an extent resulting in a level of simplification with only straight lines and no polylines. We term the straight lines *axes* (based on the terminology for scales in plant morphology [67]). The subgraph representing the branching structure with only *axes* has a set of nodes, each representing one *axis*, and a set of branching edges. Each branching edge connection between a pair of *axis* nodes represents a *ramification* as described in section 5.1.3.

To encapsulate a branching structure in a single unit, an *encompassing node* is used as a coarse representation of all *axes*. The *encompassing node* has a refinement edge to each *axis* representing node. Figure 5.7 illustrates a multiscale graph representation of a branching structure, including multiple levels of simplification, *axes*, and the *encompassing node*.

5.2.2 Global - The Multiscale Scene Graph

In this subsection, we describe a scene graph compatible with the graph representation of multi-resolution branching structures. Local graphs representing branching structures are embedded in the global scene graph as subgraphs. The scene graph takes the form of a directed acyclic graph (DAG) and refinement edges are used for connections. As in the case of conventional scene graphs, nodes representing 3D transformation can be included, usually as turtle commands.

Generating the scene graph: We use rule statements in XL to construct a scene graph. For example, the initial rule

```
Axiom ==> for(int i:(1:5))(
[ /> Translate(random(0,10), random(0,10),
 random(0,10)) /> P /> Axis /> Sim /> Tip ])
```

generates five seminal branching structures at random positions. Figure 5.8 depicts the graph along with annotations segregating the local and global aspects. $Translate$ represents a 3D translation in the scene. It is not part of any branching structure and belongs to the global aspect. P is the *encompassing node*, $Axis$ is the *axis*, Sim is the simplification line, and Tip is the initial branch tip for a branching structure. Together, these three modules are the local aspect of the multiscale scene graph. By connecting nodes in

the global aspect to *encompassing nodes* of branching structures, the aspects are bridged and we arrive at an integrated multiscale scene graph. Several edge connections to *encompassing nodes* are used for instanced branching structures (Figure 5.8, right side).

5.2.3 Update and Extraction for Rendering

Cache and error estimation updates: Each graph node representing a part of a branching structure is associated with a set of cache values containing position and orientation information in the form of a matrix. The cache contains additional information for nodes representing simplified polylines or ramification *axes*. If a node represents a simplified polyline, the length of the simplified line and the polyline simplification error is stored in cache. If a node represents an *axis*, the ramification simplification error (for which the *axis* is the child ramification line) is stored in cache. An axis-aligned bounding box is cached for *encompassing nodes*.

Cache values for modified graph nodes, excluding newly created nodes, are set as *dirty* in each simulation step. In addition, cache values for graph nodes that belong in the same *axis* as modified nodes are likewise set as *dirty*.

Two types of graph traversal in the local aspect of the scene graph update the cache and simplification errors of branching structures. Figure 5.9 shows the pseudo-code for the first type of traversal that updates position, orientation, length, and polyline simplification error. The second type of traversal, shown as pseudo-code in Figure 5.10, updates ramification simplification errors for *axes*, and axis-aligned bounding boxes for whole branching structures.

Extraction traversal: Once cache values and error estimations are updated, the scene graph is ready to be traversed for rendering. Traversal begins from the root node of the multiscale scene graph and proceeds depth-first, following directed paths composed by refinement edges. View-dependent LOD extraction begins when traversal reaches the *encompassing nodes* of branching structures.

Upon reaching an *encompassing node*, the distance d in the viewing direction from the eye point to the axis-aligned bounding box of the branching structure is computed. If d is less than a specified distance threshold, traversal continues until graph nodes representing the unsimplified structure are encountered and rendered based on their positions, orientations, and lengths. In this case, nodes representing simplifications are ignored.

On the other hand, traversal proceeds selectively if d is more than or equal to the distance threshold. The quadratic object-space geometric error threshold ϵ^2 is computed for the branching structure encountered. The linear object-space geometric error [35] threshold is computed by

$$\epsilon = p\frac{2d\tan\frac{\theta}{2}}{x} \tag{5.8}$$

where p is the screen-space geometric error threshold in pixels, θ is the field of view, and x is the resolution of the screen in pixels. We illustrate the selective traversal in Figure 5.11.

5.3 Implementation and Results

We implement the proposed methods on a Pentium 4 2.6 GHz computer with 4 Gb of memory. 3D visualization is implemented using OpenGL. Our experiment consists of empirical measurements made on simulated juvenile trees.

The methods in Section 5.1 and 5.2 are used to simulate the development of a tree (a branching structure) for 40 time steps. Figure 5.12 shows the number of axes, line segments, and time taken to compute simplification errors at each time step. Computation time is relatively linear in relation to time steps.

Figure 5.13 shows the tree at various linear object-space geometric error threshold values. Polyline simplifications and ramification simplifications operate concurrently, effectively reducing the branching structure to only a few main axes at high thresholds.

5.4 Summary and Future Work

We presented a new data structure and algorithms for level of detail of branching structures. Our examples show instanced structures generated procedurally by multiscale rules. To our knowledge, our approach is the first to consider LOD, a key component of real-time graphs, for branching structures that develop. We presented experiments that demonstrate the viability of such an approach, suitable as an extension to L-system based models for visualization.

Some interesting modifications in the future are:

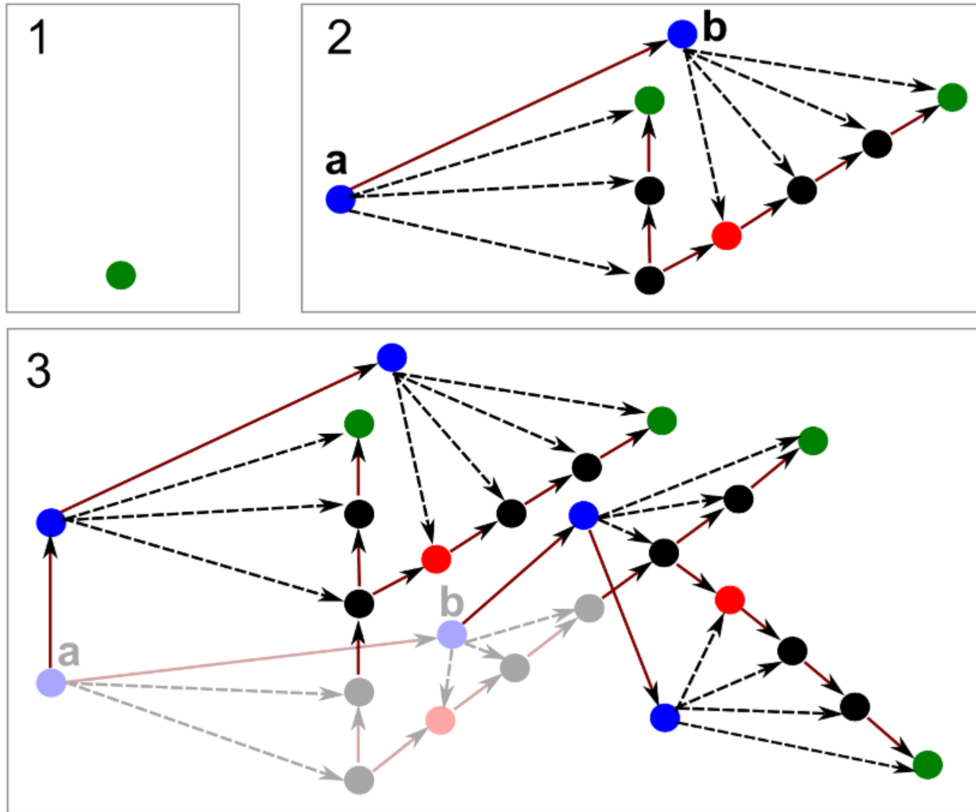
Non-branching structures: Most scenes include non-branching structures and objects. A seamless integration of our approach and data structure with other LOD computation methods for these objects would be useful to allow collective visualization.

Parallel computation: Despite the incremental algorithm, more efficiency may be achievable if computations are performed in parallel. One possible way is to perform graph traversals in parallel when updating simplification errors.

Lines with varying thickness: Some branching structures have line segments that have varying thickness. The most common example would be plants. Increased accuracy of our algorithms for these structures to take into account line thickness is desirable.

Dynamic deformation: Branching structures deform with time. For example, parts of the structure may bend further over time under the influence of external forces. Research towards an incremental approach to compute the LOD of such structures is another interesting area for future work.

Remark: The content in this chapter was submitted [127] for publication but are under modifications after review.




```
Tip ==> Sim F [Sim RU F F Tip] F Tip;
```

Figure 5.6: Applying the multiscale rule for 2 time steps. Green nodes are *Tip* nodes, black nodes are *F* nodes, red nodes are *RU* nodes, and blue nodes are *Sim* nodes. Black, dotted edges are refinement edges. Colored, solid edges are successor or branching edges. The multiscale rule is first applied to the single *Tip* node in Box 1 to derive the graph in Box 2. The next application derives the graph in Box 3 from the graph in Box 2. The *Sim* nodes persistent in the second rule application are labelled **a** and **b**. Faded nodes and edges in Box 3 are created in the earlier (first) rule application, not in the second rule application. The subgraph consisting of all blue *Sim* nodes and colored edges between them is one level of simplification for the unsimplified structure.

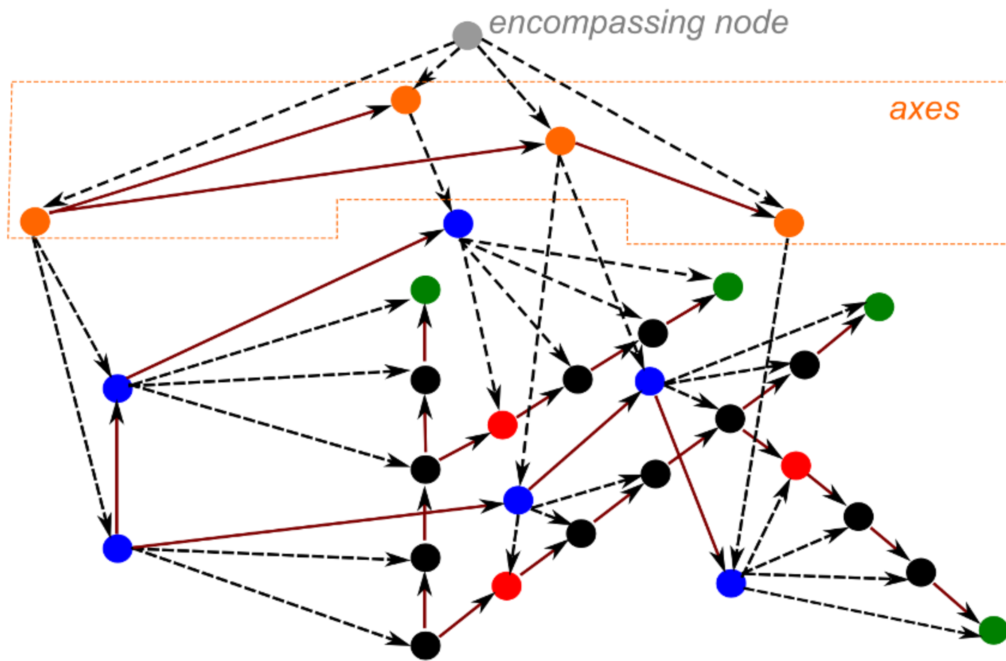


Figure 5.7: Multiscale graph representation of complete branching structure. The grey encompassing node refines to the axis-representing nodes in orange. The dotted orange boundary highlights the level of simplification made up by axes. The pattern and coloring of edges and nodes are consistent with that in Figure 5.6.

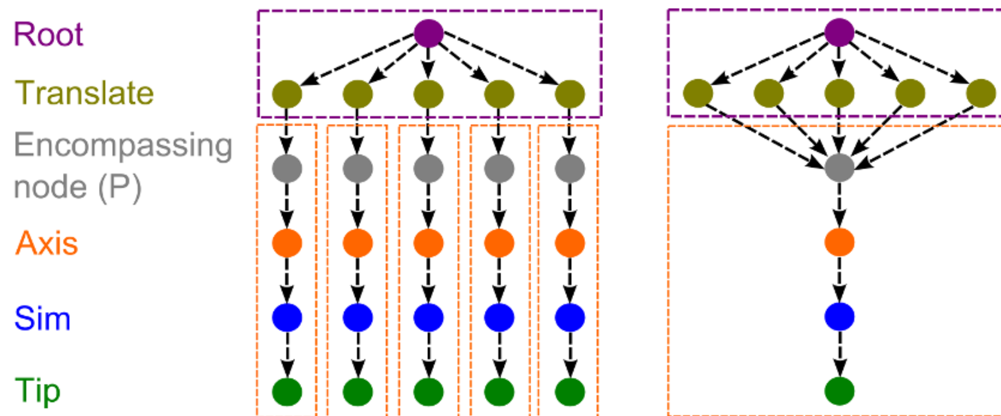


Figure 5.8: Illustrations of scene graphs. Node type names are listed on the left with the same colors as the nodes they represent. Black, dotted edges are refinement edges. Left: Five branching structures randomly positioned using $Translate$ nodes. Right: Five branching structures placed in scene by establishing edge connections to a single instanced branching structure. The global aspects of the scene graph are highlighted using dotted, purple boundaries. The local aspects are highlighted using dotted, orange boundaries.

```
for each branching structure instance
  traverse1(root node of unsimplified branching structure);

traverse1(sNode):
  if sNode is new or dirty
    set sNode position & orientation in cache;
    for each polyline simplification of sNode
      if sNode is polyline first vertex
        set position of simplification in cache;
      if sNode is polyline last vertex
        set length & orientation of simplification in cache;
      set error of simplification incrementally in cache;
  for each tNode connected an by outgoing
  successor or branching edge from sNode
    traverse1(tNode);
```

Figure 5.9: Pseudo-code for the first type of graph traversal, visiting nodes representing branching structures. The position, orientation, and length of each line segment, including polyline simplifications for multiple levels of simplification, as well as axes, are updated. Polyline simplification errors are updated using the incremental approach in section 5.1.

```
for each branching structure instance
  traverse2(root node of axes);

traverse2(sNode):
  if sNode is new or dirty
    set ramification error in cache;
  update bounding box in encompassing node;
  for each tNode connected an by outgoing
  successor or branching edge from sNode
    traverse2(tNode);
```

Figure 5.10: Pseudo-code for the second type of graph traversal, visiting nodes representing the axes and encompassing nodes of branching structures. Ramification errors are updated using the approach in section 5.1.

```
for each aNode representing an axis
   traverse3(aNode);

traverse3(sNode):
   err = simplification error of sNode
   if (err < ε²)
     if sNode is not an axis
       draw sNode;
     skip the next recursive calls;
   for each tNode connected an by outgoing
   refinement edge from sNode
     traverse3(tNode);
```

Figure 5.11: Pseudo-code for selective traversal of branching structure. If the quadratic ramification or polyline simplification error is less than the quadratic object-space geometric error threshold ϵ^2, the current sNode is an acceptable simplification and its refinements are excluded from traversal.

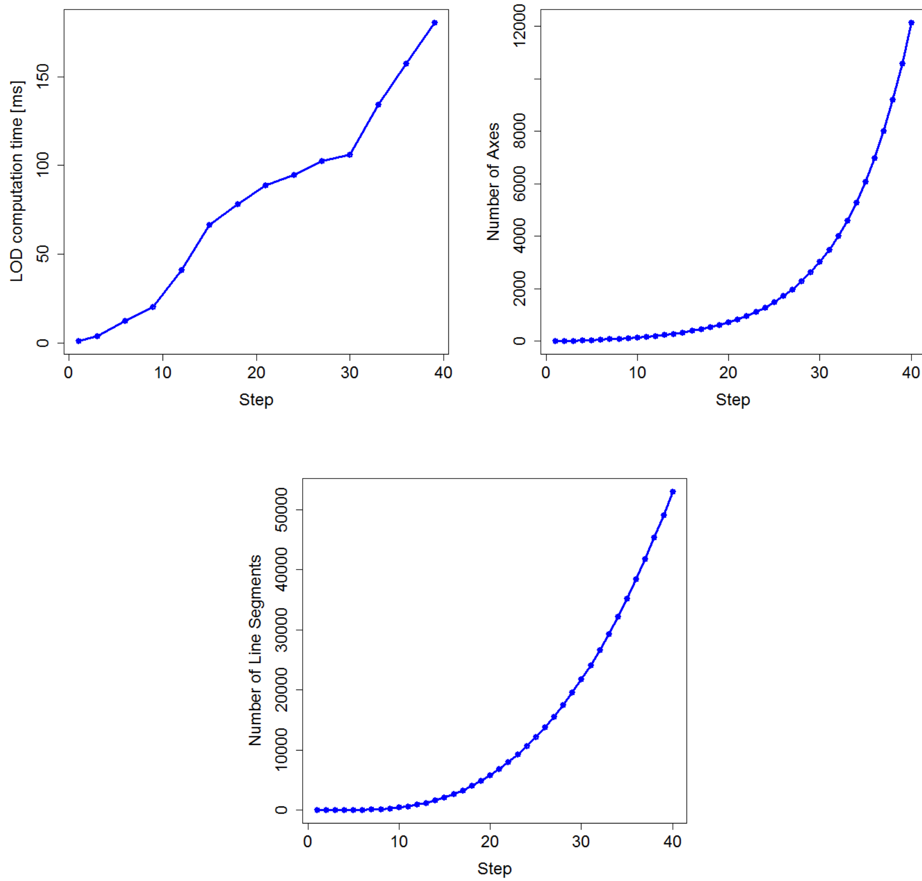


Figure 5.12: Results of simulation tree growth for 40 time steps. Top left: LOD computation time, accounting for graph traversals, polyline simplification errors, and ramification errors. Top right: Number of axes at each time step. Bottom: Number of unsimplified line segments at each time step.

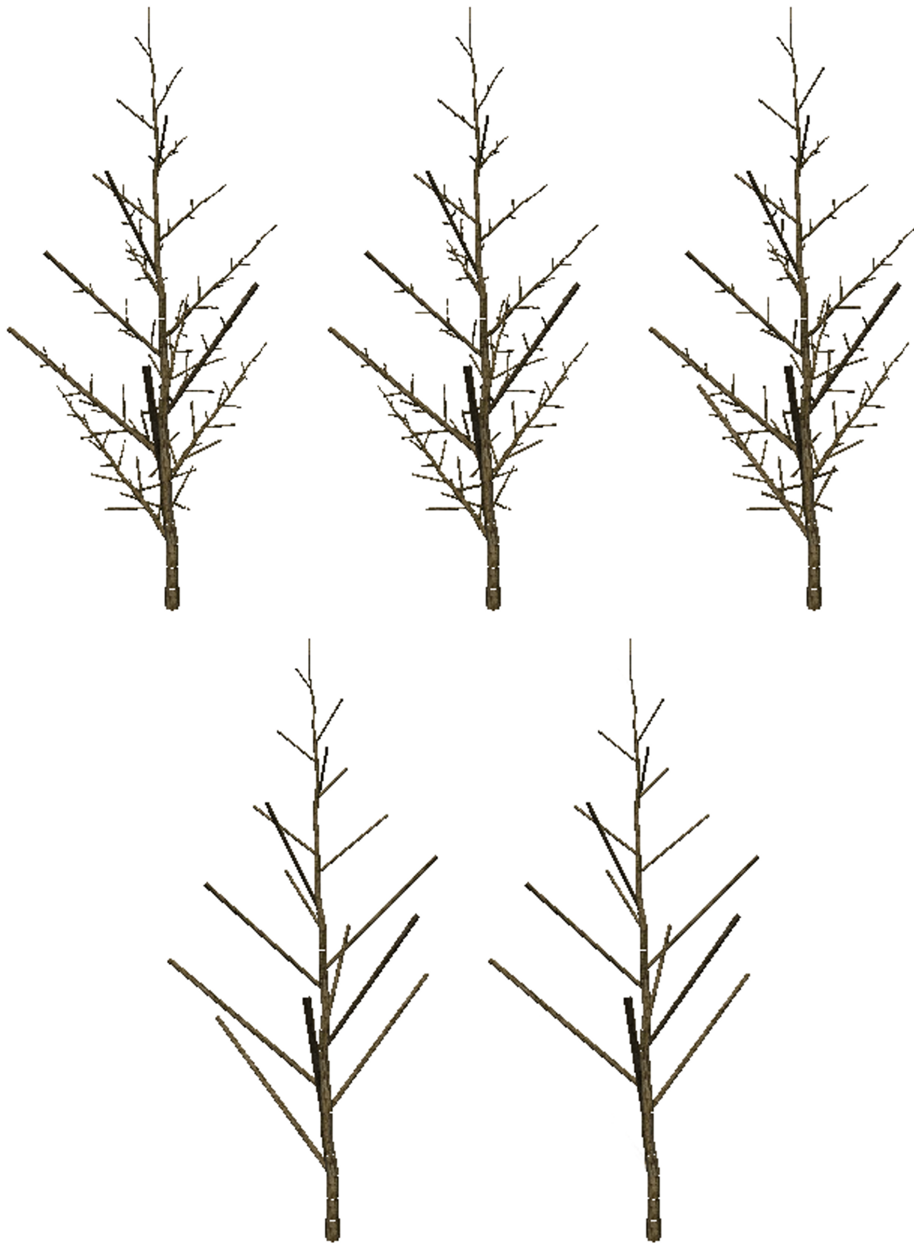


Figure 5.13: Multiple resolutions of branching structure. Top left: ϵ at 0.002 with 1077 line segments and 898 axes. Top center: ϵ at 0.014 with 607 line segments and 124 axes. Top right: ϵ at 0.058 with 150 line segments and 124 axes. Bottom left: ϵ at 0.233 with 31 line segments and 18 axes. Bottom right: ϵ at 0.5 with 0 line segments and 15 axes.

Part III

Applications & Examples

Chapter 6

Examples and Demonstrative Models

This chapter contains examples and demonstrative models for the concepts and techniques described in this thesis. The sections contain information published in [128], [130], and [129]. While some examples contain isolated conclusions, collated conclusions referencing several examples together are presented in Chapter 7.

6.1 Fission Yeast

A multi-cellular model of the fission yeast (*Schizosaccharomyces pombe*) cell division is implemented. The model and simulation parameters are based on the multi-level approach by Maus et al. [115]. However, our multiscale graph grammar approach allows arbitrary functions and rules to be applied to any part of the model data. Multiscale causation in our graph model is illustrated and visualized by interpreting the graph as a scene graph.

6.1.1 Single Cell Model

The yeast cell cycle consists of four phases: G_1, S, G_2 and M. During the phases G_1, S and G_2, the cell increases in size. Following the G_2 phase, the cell enters the M (mitosis) phase and divides into two daughter cells. Phase changes are regulated by two proteins - cyclin and cdc2, that form a complex called maturation promoting factor (MPF). MPF controls the

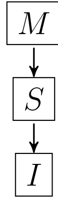


Figure 6.1: Fission yeast cell division simulation structure of scales. S is a scale representing cells and lists of cells. M is a scale representing grid space and sub-divided space (tiles). I is a scale representing proteins and complexes at a micro level.

traversal of cell cycles. This protein regulation model is based on an early model by Tyson [172]. Fission yeast cells may undergo sexual reproduction when environmental conditions are getting poor. The mating types (P and M) of cells enforce fusion of opposite types [100]. Fused diploid zygotes undergo sporulation and later germinate to create haploid cells. The mating type of proliferating cells switches sporadically [177] and can be observed by phenomenological patterns [90]. We define cells as a module *Cell* in XL with the attributes: *volume*, *state*, *mateType*, *switchable*, x and y. *state* represents the cell cycle phase, *mateType* represents either the P or M mating type and *switchable* represents the phenomenological type for determining the mating type of spawned cells.

Proteins and complexes (cyclin, phosphorylated cyclin, cdc2, inactive MPF, active MPF and repressed MPF) are also defined as modules (named Y, Yp, D, MI, MA and MR respectively) extending from a base module representing species in general. The base module consists of an attribute for molecule count. Proteins and complexes exist as nodes in the model graph. A node labelled *CellList* represents a collection of cells. Each *Cell* node is connected from the *CellList* node by a branching edge. Each cell is refined to its intracellular components represented by nodes of the protein or complex modules. Proteins or complexes taking part in the same intra-cellular processes are connected in a chain of successor or branching edges. Figure 6.2 shows an example of the graph model and Figure 6.1 shows the corresponding structure of scales.

Rules are defined modelling the intra-cellular processes. One such rule, illustrating an inter-scale causation is:

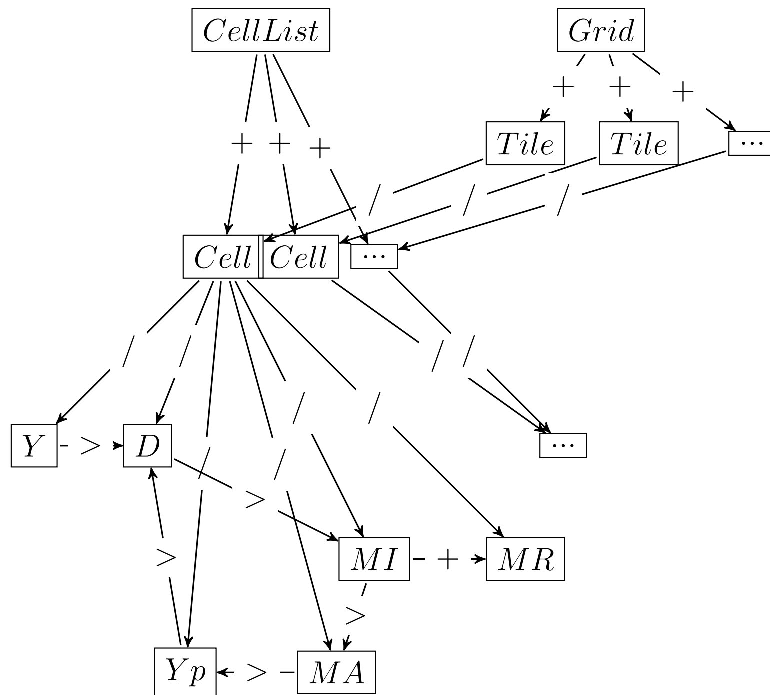


Figure 6.2: Fission yeast cell division simulation model graph. *CellList* and *Cell* nodes belong to scale S, *Grid* and *Tile* nodes belong to scale M and the other nodes belong to scale I (scales seen in Figure 6.1).

```
c:Cell ma:MA yp:Yp d:D ::>
{ double r = (k4/c[volume])*ma[mol];
    ma[mol]=ma[mol]-r;
    yp[mol]=yp[mol]+r;
    d[mol]=d[mol]+r; ...  }
```

The volume of the cell is used along with the amount of active MPF to determine a reduction in the amount of active MPF in the cell. '::>' is a rule for code execution without changes to the graph structure. $k4$ is a constant value. Notice that MA, Yp and D are connected sequentially by successor edges, simplifying the query statement in the rule, i.e. no explicit expressions of the edges are required. Advanced implementations with ordinary differential equations in GroIMP [75] can also be used instead.

The model graph functions additionally as a scene graph. The *Cell* module extends the *Sphere* module and is therefore graphically rendered in the 3D-view of GroIMP as a sphere. Graph nodes contain transformation information (e.g. rotation, translation, scaling) used during the traversal of the graph for rendering. The cell nodes in this case contain translation information for their positioning on a two-dimensional plane using their x and y attribute values. Entities from the finest scale, i.e. proteins and complexes, are not visualized in this example.

6.1.2 Multiple Cell Model

The fusion of fission yeast cells is regulated by pheromone molecules secreted by the cells. Cells of mating type P secrete P-factor pheromones and cells of mating type M secrete M-factor pheromones. Cells sense the pheromones secreted by the opposite mating type, causing an arrest at their G_1 cell cycle phase [163]. Cells of mating type M also secrete a P-factor-specific protease ($Sxa2$) that reduces the effect of P-factor pheromones. This process is modelled using a two-dimensional grid space where each sub-divided grid area (a tile) contains information about the quantity of both types of pheromone and protease $Sxa2$ in the tile. In our implementation, the entire collection of tiles is defined by a module *Grid*. Each sub-divided space is defined by a module *Tile* with attributes: x, y, Fp, Fm and $Sxa2$.

x and y are the positions of the tile on a two-dimensional plane, Fp is the amount of P-factor pheromone, Fm is the amount of M-factor pheromone and $Sxa2$ the amount of protease $Sxa2$ in the area represented by the tile.

Each tile is connected to the cell nodes that are residing in its spatial territory, i.e. the 1 unit^2 square area from (x,y) to (x+1, y+1). Figure 6.2 shows the model with the *Grid* node representing the entire grid space, *Tile* nodes for each sub-divided area and refinement connections to individual cells residing in the respective tiles. The cell cycle arrests are dependent on the amount of pheromones in the containing tiles of the cells. Pheromones also diffuse from tile to tile in four directions (up, down, left and right). The following rule spanning 3 scales models the conversion of inactive MPF to repressed MPF depending on tile pheromone quantity, causing the arrest of cell cycles:

```
v:Tile c:Cell mi:MI [mr:MR] ::>
{ ...
    if(c[mateType]==typeM)
       rate = (k11*(v[Fp]**3))/
          (K11+(v[Fp]**3));
    else
       rate = (k11*(v[Fm]**3))/
          (K11+(v[Fm]**3));
    rate = rate/(c[volume]**2)*mi[mol];
    mi[mol]=mi[mol]-rate;
    mr[mol]=mr[mol]+rate; }
```

The following rule models the spawning of new cells and their connection to both the *CellList* node and *Tile* node. The details of copying and setting the attribute values from the parent cell to the new cell are left out intentionally. It is our intention here to demonstrate the rule-based modification of the graph structure. A new cell and a new set of proteins and complexes are added using the production statement on the right-hand side of the rule.

```
g:Grid [v:Tile c:Cell y:Y d:D mi:MI [mr:MR] ma:MA yp:Yp],
clist:CellList [c],(c[state]==M),
(ma[mol]<t9)
   ==>
g [v c y d mi [mr] ma yp]
[v nc:Cell ny:Y nd:D nmi:MI [nmr:MR] nma:MA nyp:Yp],
clist [c] [nc];
```

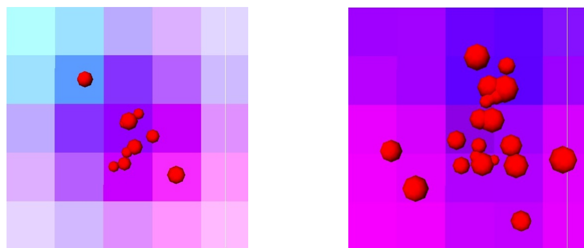


Figure 6.3: Screenshot of cell division process visualization. The white circles are cells. There are 25 tiles. Color intensity indicates strong presence of pheromone in the tiles. The sizes of the circles depend on the cell volumes. The picture on the left shows an earlier simulation state compared to that shown on the right.

The *Tile* module is defined such that it extends the graphically meaningful *Parallelogram* module and contains translation information to the respective grid positions.

Rules for cell movement dynamics to avoid crowding of cells in tiles are implemented. When cells move into the space of another tile, the refinement relation is updated accordingly. Figure 6.3 shows the top-down three-dimensional visualization of the simulation in GroIMP.

6.1.3 Rule-based Species and Complexes

Since the above defined set of proteins and complexes is contained in each cell, it is questionable why they are not defined as attributes of the Cell module. We would like to illustrate the possibility of creating complex species by applying rules to a basic set of species. Figure 6.4 shows four interacting protein species represented by nodes refined from a *SPool* (species pool) node. Using rules, we can create complex species that combine the basic species as nodes of a coarser scale. Complex species nodes refine to the basic species nodes and are added into a pool of complex species *CPool*. Figure 6.4 shows the creation of a few complex species. This approach of generating possibly large numbers of combinatorial complexes is introduced in [115]. Although our newly introduced graph model is suitable for this purpose, the functionality for translating rule-generated complexes and their reactions into attributes and functions of modules in XL is work-in-progress as part of a component-based modelling project in our team. In this yeast model

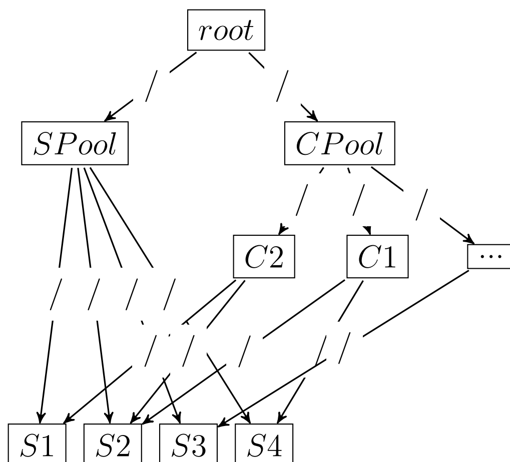


Figure 6.4: Species and complexes. Rules can be used to define a large number of combinatorial complexes. $SPool$ is a pool of species, $CPool$ is a pool of complexes, S labelled nodes are species and C labelled nodes are complexes.

example, we justify our choice of using nodes instead of module attributes to make conscious the possibility of using graph nodes to represent protein and complex species as well as the multi-scale causality with cell nodes.

6.2 Beech Structural Growth

The continued virtual growth of a measured young European beech (*Fagus sylvatica*) is simulated using the multiscale graph model and grammar. Canopy height and topological scales are combined in a single simulation model. The branching structure is represented as internodes at the finest scale. Two coarse scale representations of internodes are used: annual growth units and height layers. The corresponding structure of scales is shown in Figure 6.5.

The graph structure of growth units and internodes is similar to that in Multiscale Tree Graphs [67]. Internodes are represented as graph nodes connected using successor and branching edges. Growth units are also represented as graph nodes connected in the same way. Internode nodes are connected to growth unit nodes that they belong to using refinement edges. Statistical information, e.g. the number of internodes per growth unit, can

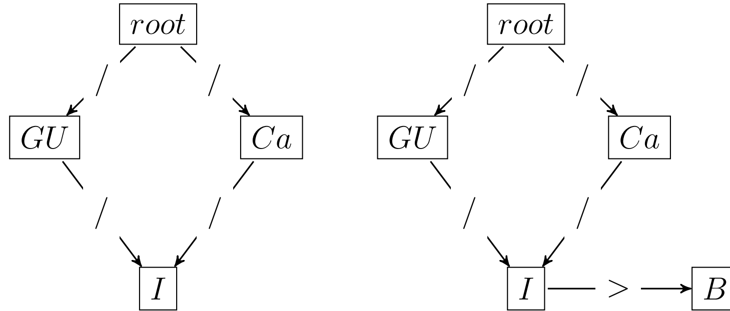


Figure 6.5: Beech model structure of scales and type graph. Left: GU is a scale representing growth units, Ca is a scale representing height layers, and I is a scale representing internodes. Right: Type graph similar to the structure of scales but with additional node B.

be utilized for the basic simulation of structural growth.

The percentage of above canopy light (PACL) is unevenly distributed in tree crown layers. Light interception in relation to canopy height is commonly explored [154] and related to senescence and mortality [135] of plants. To model self-pruning, we represent every interval of height above ground with a Ca (canopy) node. Internodes are connected to these Ca nodes with refinement edges based on their height above ground. For consistency, we use the top position of each internode for the categorization.

At the start of the simulation, the measured information for a young beech tree is input using XL. Height layers are created, each representing a hundred millimetres, up to three metres. Heights for the initial internodes are computed and connected with the respective Ca nodes. The type graph used is shown in Figure 6.5.

Each simulation step creates a number of internodes and corresponding growth units. For the demonstration of this graph model, we use a simple random number of internodes from one to three. The length of the newly created internodes decreases exponentially with the order of the branch. Figure 6.6 shows the graph rewriting step simulating structural growth.

Self-pruning of the tree is simulated using the height layer scale. Probability of pruning an internode exponentially decreases with height. Figure 6.7 shows the visualization results. This virtual tree growth example illustrates the use of the multiscale graph model and grammar for multiscale modelling in functional-structural plant modelling.

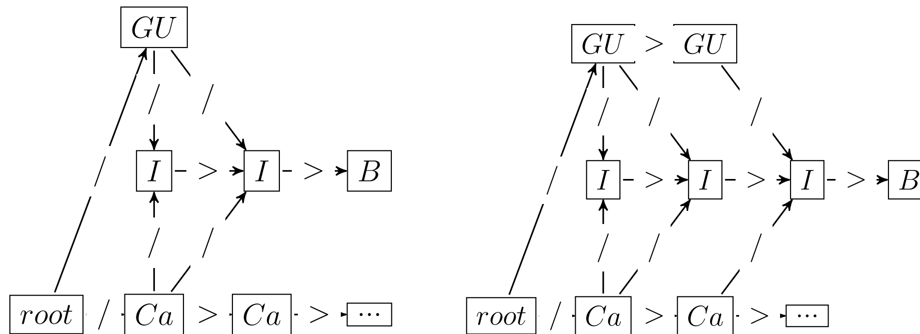


Figure 6.6: Beech structural growth. GU represents a growth unit node. I represents an internode node. Ca represents a height layer node. B represents a meristem tip of a branch. The graph on the left shows the structure before rule execution. The graph on the right shows structural growth after a single rule execution. An additional internode belonging to a new growth unit is added and connected to the next height layer.

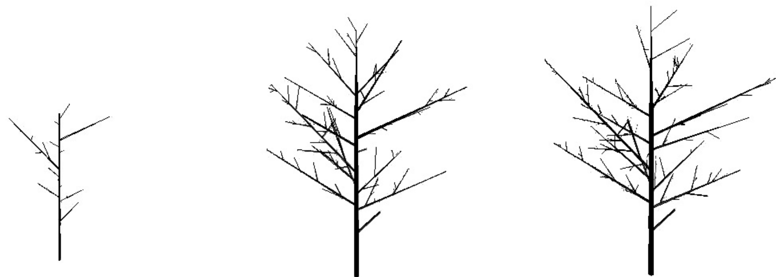


Figure 6.7: Beech structural growth. The left-most image shows the measured tree before simulation. The middle image shows the result of simulated growth with pruning using height layers. The right image shows the result of simulated growth without pruning.

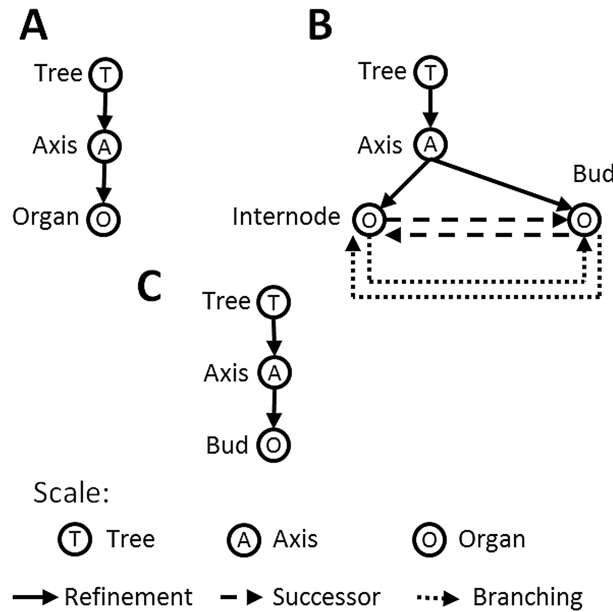


Figure 6.8: An illustration of the structure-of-scales (A), type graph (B) and initial instanced graph (C) for modelling a multiscale plant structure.

6.3 Specifying and Generating a Multiscale Plant Structure

A structure-of-scales (Figure 6.8A) is made to represent the tree, axis and organ scales of a tree. It is not explicitly declared in source code, but can be derived from the type graph. The model's type graph (Figure 6.8B) consists of types Tree, Axis, Internode, and Bud. The Internode and Bud types belong to the organ scale while the Axis and Tree types belong to the axis and tree scales respectively. The types in the organ scale are pairwise inter-connected by branching and successor edges to allow these relationships between them.

The type graph is constructed in XL as shown in Listing 6.1.

```
==>> ^ /> TypeRoot /> Tree
       /> Axis
       /> {# Internode Bud #};
```

Listing 6.1: XL Type Graph Construction

Because nodes at the organ scale (i.e. Internode, Bud) can be connected to one another by successor or branching edges, a clique (complete graph; syntax {# ... #}) is established for these node types in the type graph such that the pair is completely (in graph terminology) inter-connected by branching and successor edges (cf. Figure 6.8B).

To initialize simulation, the model's instanced graph is created with a Tree node refined to an Axis node that is further refined to a Bud node (Figure 6.8C):

```
Axiom ==> Tree /> Axis /> Bud;
```

Each simulation step consists of parallel applications of a rule that replaces a bud with a new internode, lateral axis, lateral bud and apical bud:

```
Bud ==> Internode [Axis Bud] Bud;
```

`Axis` is the only node that belongs to a coarser scale in this rule. Without it, the rule appears like classical (single-scale) L-system rules. When a bud in the instanced graph is matched to the left-hand side of the rule, the query recognizes its axis and tree encoarsements at the same time, following the relationships specified in the type graph. Subsequently, organ scale nodes produced by the right-hand side of the rule, except for the lateral axis and bud, are automatically refined from the same encoarsements. For the new lateral `Axis` specified in the right-hand-side production statement, the framework establishes a branching edge from the existing parent axis node and a refinement edge from the tree node automatically. The new lateral `Bud` branches from the newly produced internode and refines from the new lateral Axis. Figure 6.9 illustrates the modification of the initial instanced graph after one application of this rule. A detailed account of grammar operations at multiple scales is given in Chapter 4.4 and in [128]. The resultant data structure is a multiscale graph (in this case, also an MTG) representing the plant topology.

This example generates a multiscale representation of plant topology. We have implemented an alternative model with single-scale rules to achieve the same goal (see supplementary material in [130]). In this model, coarse representations (not *scales* since the model is not formulated with a structure-of-scales or equivalent) of organs are implemented using an object-oriented (O-O) approach. To represent the axes and the organs in O-O code, a Java

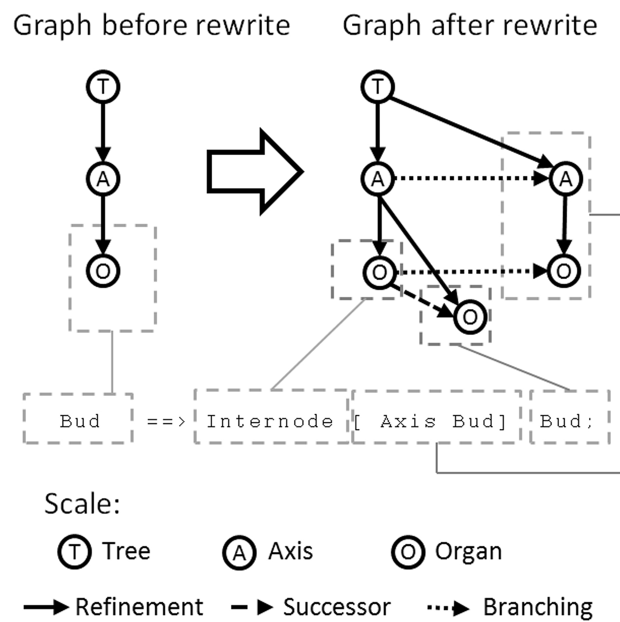


Figure 6.9: An illustration of the initial instanced graph modified by an XL rule. The initial Bud node is removed and replaced by a new internode, lateral axis, lateral bud and apical bud.

class `Axis` and a Java interface `Organ` are implemented in separate files. An `Axis` object has references to other `Axis` objects to which branching relationships exist, as well as references to its fine representations at organ scale. In the main modelling file, an array is defined to contain all axes in the plant. XL modules for internodes and buds are defined with an index reference to the array, identifying their respective encoarsements:

```
module Internode(int axis) implements Organ;
module Bud(int axis) implements Organ;
```

The rule to generate the same multiscale structure from this example is then specified as:

```
b:Bud ==> {
    removeOrganFromAxis(b.axis, b);
    int latAxis = createAxis(b.axis);
}
i:Internode(b.axis, 1)
[bl:Bud(latAxis)]
ba:Bud(b.axis)
{addOrganToAxis(b.axis, i); ...}
;
```

The production statement is sandwiched between object-oriented code blocks (between curly braces) that invoke methods to maintain and create refinement relationships between axes and organs. The method `removeOrganFromAxis` is invoked to remove the matched Bud node from its axis encoarsement. After the specification of the production graph (consisting of a new internode, lateral bud and apical bud), the method `addOrganToAxis` is invoked multiple times to add references of the newly created organs into the axis objects.

By comparing this implementation with this example, some advantages of the three-part multiscale graph model and grammar are identified. One significant advantage of the multiscale rules is the implicit handling of refinement relationships once a type graph is specified. An index or reference-based implementation of scales together with classical single-scale rules, as shown in our alternative implementation, requires the modeller to explicitly maintain scale relationships with method invocations, introducing more room for

error. Another advantage of the multiscale rules over index-based implementations is the implicit type constraint, i.e. rules referencing the type graph are less likely to generate erroneous refinement relationships. In contrast, such constraints need to be manually handled by a modeller who chooses an index-based implementation approach, since indices are not programmatically tied to specific arrays. Comparing the length of the implementation code files reveals a third advantage of the multiscale rules. For generating a simple multiscale structure, multiscale rules require less than 30% of the code (ignoring empty and commentary lines) required by single-scale rules.

6.4 Crown Generation

In this example, we demonstrate the dependency of an artificial light-sensitive tree on fine scale organ developments. Light sources are directed at a growing tree from equal distances vertically above, diagonally around from an elevated height and horizontally around from ground level. The model is catered to produce coarse-scale outputs such as tree height and crown dimensions (e.g. radii in different directions). These outputs are derived from organ developments extrapolated from finer to coarser spatial and time scales.

The structure-of-scales in this example has a tree scale refined to an organ scale (Figure 6.10A). Several types defined in the type graph (Figure 6.10B) make up the model of the growing tree. At the tree scale, a CrownLayer type represents a vertical quartile of the tree crown and a Trunk type represents the growing tree trunk. At the organ scale, a Bud type represents buds, a Branch type represents branch segments and a Foliage type emulates leaves to reduce light intensity. In addition, the organ scale has a Marker type to mark initial bud sample positions in crown layers. The types at the organ scale are inter-connected by branching and successor edges to allow these relationships between them. An instanced graph (Figure 6.10C) is created with one node for the trunk and four nodes for the crown layers of the growing tree. Light sources are included as nodes in the graph but are excluded from figures for simplicity. The XL code for creating the type graph is similar to that in the previous example.

Tree growth is modelled by a series of simulation steps. In each simulation step, crown growth is formulated as a multiscale problem (see Section 4.1 on the *problem categorization*, *scale dependency*, and *scale integration* components of the multiscale modelling framework) where the tree scale is entirely

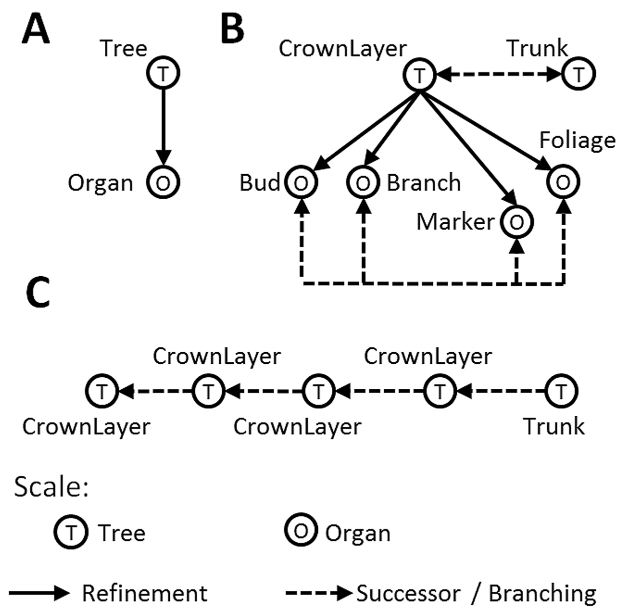


Figure 6.10: An illustration of the structure-of-scales (A), type graph (B) and initial instanced graph (C) for modelling the growth of four crown layers based on the development of a sample of organs.

<table>
<tr><td colspan="2">Problem Category</td><td>Crown growth always based on bud and branch developments.</td></tr>
<tr><td rowspan="2">Dependency</td><td>Dependent</td><td>Tree (crown layers)</td></tr>
<tr><td>Independent</td><td>Organ (branches and buds)</td></tr>
<tr><td rowspan="3">Integration</td><td>Initialization</td><td><u>Bud positioning</u> - Based on vertical distribution and crown dimensions, a representative number of buds are positioned.</td></tr>
<tr><td>Evolution</td><td><u>Branch growth</u> - Rule-based branch elongations over short time scale.</td></tr>
<tr><td>Extrapolation</td><td><u>Crown estimation</u> - Bounding boxes of branches scaled up and aggregated into new crown dimensions.</td></tr>
</table>

Table 6.1: Multiscale problem categorization, scale dependency and scale integration for example in Section 6.4 (crown generation).

scale dependent on the organ scale, i.e. crown dimensions are determined by the developments of light-sensitive buds and branches. An overview of the scale integration, in this case also a Monte Carlo method [84] application, is shown in Table 6.1.

In *initialization*, the age of the tree determines the number of buds. From the finite population of buds, a representative sample is created. A wide range of sampling methods can be used. In this example, simple random sampling with finite population correction [105] is employed. The mean number of the buds forming the height and maximum radii in each direction of the crown layers is assumed to be 20% of the bud population. The sample buds are independently distributed to each crown layer following proportions given as inputs to the model. Each sample bud is assigned a random position within its crown layer (Figure 6.11A). The sampled and positioned buds are refined from their respective crown layers. For example, the XL code to establish the refinement is: `cl [createBud(cl)];` where `createBud(cl)` is a method returning a new Marker graph node representing a single bud from the samples in the crown layer represented by `cl`. The two nodes are connected by a refinement edge although `createBud` is contained within

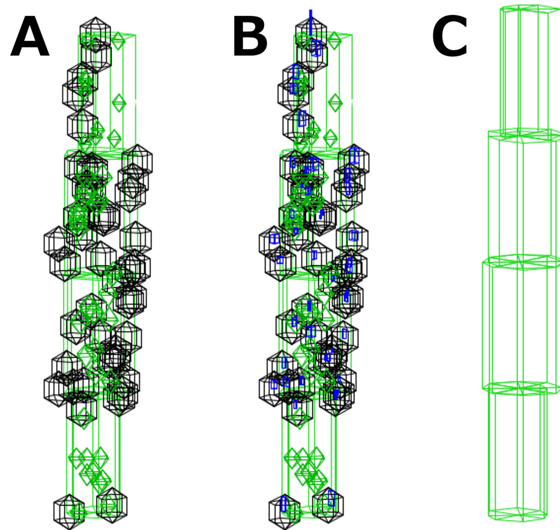


Figure 6.11: Illustration of multiscale crown model development. Black spheres represent samples of buds (sizes intentionally increased for visibility). Green spheres represent random foliage. Green frustums represent crown layers. (A) Random positioning of bud samples and foliage spheres in initialization. (B) Up-scaled bounding boxes of fine organs shown in blue. (C) Result of extrapolation. Crown layers are updated to reflect new tree height and crown dimensions containing the bounding boxes in (B).

square brackets because of the refinement relationship defined in the type graph. Spheres are placed in each crown layer to emulate leaves for self-shading within the crown (similar to the approach by [26]). The volumes of these spheres are proportional to the volume of their respective crown layer. The rule-based code for refining foliage graph nodes from crown layer nodes is similar to the refinement of crown layers to buds.

In *evolution*, a photon-tracing technique [76] is used to obtain irradiance of each sampled bud from each light source. The final growth direction of branches from each bud is determined by a weighted sum of the direction towards the light source providing maximum irradiance and a default direction for the crown layer given as model input (similar to the method by [133] but tropism is accounted for in the default direction). Rules generate branch segments for a fine-scale (i.e. short) time frame. Figure 6.12 illustrates an example of the graph modifications when rules are executed

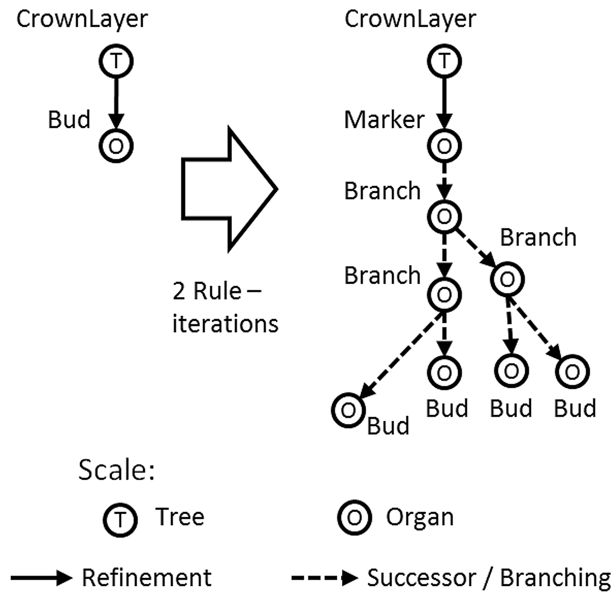


Figure 6.12: Modification of instanced graph at organ scale. After two iterations, the bud produces three branches. The branches with paths to the same marker node are eventually used to create a bounding box at the organ scale.

to create branch segments from buds. In *extrapolation*, minimal bounding boxes enclose branches originating from each Marker object. The dimensions of these bounding boxes are scaled up based on the ratio of the macro-scale simulation time to the micro-scale simulation time (Figure 6.11B). Finally, existing crown dimensions and the up-scaled bounding boxes from each bud node are aggregated into new crown dimensions (Figure 6.11C). Sample buds are discarded at the end and the three processes (initialization, evolution and extrapolation) repeat for each step.

To concretize a comparison of this example against single-scale approaches of plant modelling, we implemented a corresponding model that contains complete plant topology and light conditions by Palubicki et al. [133]. Four plant architectures with contrasting apical dominance and tropism (cf. Figure 7 and Figure 12 in [133]) are implemented using both models. Figure 6.13 shows that the model in this example, i.e. the multiscale framework, is capable of replicating selected architectural shapes with specific parameters. No crown dimensions or tree scale aggregates are created from the

single-scale implementation for comparisons because aggregates constitute dependencies that make a model multiscale in nature. A correlation-based crown development model, on the other hand, cannot offer fine light sensitivity for comparisons.

Due to the self-replicating characteristic of plants, the numbers of branches and buds can increment exponentially during the juvenile growth phase, often causing computational limitations in single-scale approaches (one of the earlier mentioned motivations of multiscale modelling). The example in this section overcomes this while retaining fine sensitivity to light by sampling the bud population. Juxtapositions of bud count and simulation time for the multiscale and single-scale implementations are shown in Figure 6.14 and Figure 6.15.

The above advantages are attributed to the multiscale framework and not to the sampling method. An attempt to utilize bud sampling for the single-scale model requires a consideration of bud positions in relation to the crown's geometric space, i.e. only buds near the boundary of the crown should be sampled, deeming the model a multiscale model. Moreover, with sampling, light model mechanisms in the single-scale model still operate for a large number of branches and buds. A salvaging attempt to use coarse representations of woody and foliage objects would constitute, once again, a multiscale model. Despite the advantages, the implementation of this example requires additional procedures which are absent in the single-scale model to up-scale organ developments to the tree crown. More specifically, the scaling and aggregation of bounding boxes to crown layer dimensions lengthens the simulation pipeline. In addition, this example produces output at tree scale, i.e. at coarse spatial resolutions, limiting its practicality to cases where fine-scale outputs can be ignored. Additional coarse-scale inputs such as the vertical distribution of buds and spatial distribution of foliage in crown layers are also not required in the single-scale approach.

6.5 *Fagus sylvatica* Stand under Ozone Exposure

European beech (*Fagus svlvatica*) is one of the most important tree species in central Europe [155]. Models and observations indicate increasing concentrations of tropospheric ozone in Europe since 1996 [41]. Tropospheric ozone

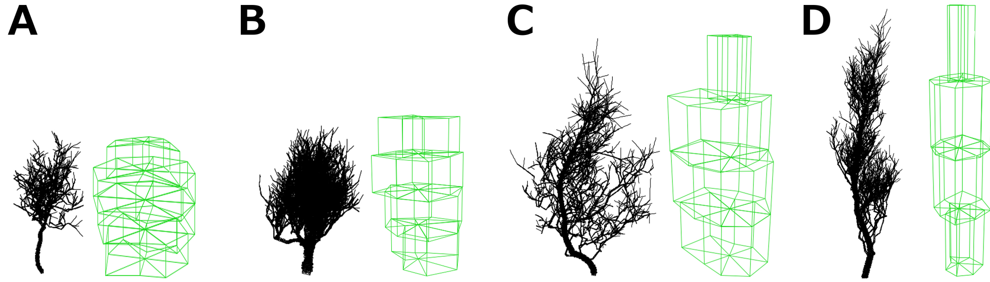


Figure 6.13: Black trees depict results of the single-scale model (SS) with complete topological data. Green crown layers show results of the corresponding multiscale model (MS) parameterized from the example in Section 6.4. Each architectural type is described with distinguishing source code parameters values. ALLOC_L and apical are parameters indicating apical dominance. PLANT_WT_TROPISM is a parameter indicating tropism in branch development. budAngleRL consists of the minimum and maximum branching angles in the four crown layers. O_LEN is a parameter for branch elongation length. (A) Low apical dominance and low tropism. SS parameters: ALLOC_L=0.49, PLANT_WT_TROPISM=0.3. MS parameters: apical=0.6, budAngleRL=0,20,30,45,40,75,50,85, O_LEN=0.071. (B) Low apical dominance and high tropism. SS parameters: ALLOC_L=0.49, PLANT_WT_TROPISM=1.0. MS parameters: apical=0.6, budAngleRL=0,20,15,25,20,30,20,30, O_LEN=0.071. (C) High apical dominance and low tropism. SS parameters: ALLOC_L=0.54, PLANT_WT_TROPISM=0.3. MS parameters: apical=0.78, budAngleRL= 0,20,30,45,40,75,50,85, O_LEN=0.2. (D) High apical dominance and high tropism. SS parameters: ALLOC_L=0.54, PLANT_WT_TROPISM=1.0. MS parameters: apical=0.78, budAngleRL= 0,20,15,25,20,30,20,30, O_LEN=0.2.

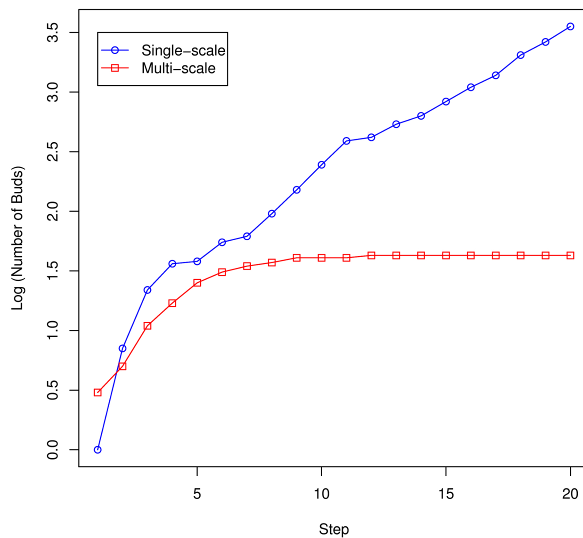


Figure 6.14: Logarithm of the number of buds in each simulation step for the single-scale model and the multiscale model (example in Section 6.4).

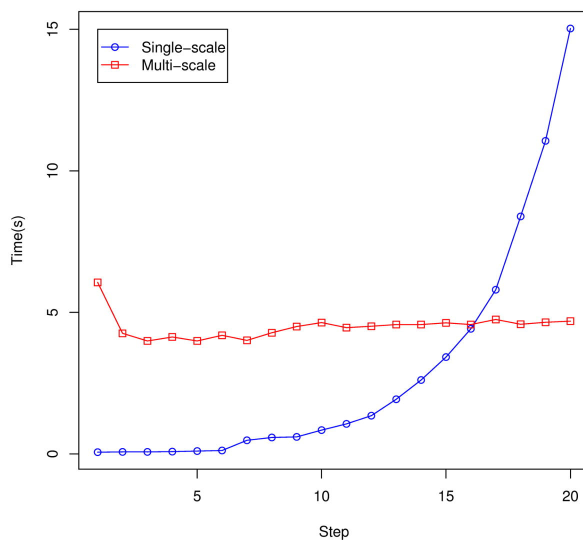


Figure 6.15: Simulation time taken at each simulation step for the single-scale model and the multiscale model (example in Section 6.4).

triggers oxidative stress responses in the enzymes of the Shikimate pathway [10] as well as in protein levels related to the Calvin cycle [88] in beech trees. These responses lead to structural (e.g. leaf lesions) and functional (e.g. photosynthetic capacity) depreciations.

A structure-of-scales (Figure 6.16A) is designed to incorporate the effects of ozone on beech tree stands. The stand scale consists of ozone concentrations and light sources. The tree scale describes the positional information of trees in a stand and aggregated attributes of individual beech trees. It is further refined into two incomparable scales: the axis scale and the crown scale. The crown scale contains collective information of tree crowns and is refined to a crown layer scale. The layers are unique vertical height sections that divide the tree crown one-dimensionally. The axis scale is refined to the growth unit scale. The crown layer scale is additionally decomposed into an organ scale that is a refinement of the growth unit scale concurrently. Lastly, the crown layer scale refines to a metabolic network scale. Here we deploy the simplifying model assumption that the metabolic network dynamics do not significantly differ between organs of the same crown layer.

The model's type graph (Figure 6.16B) illustrates the types and relationships utilized in each scale. The stand, tree, crown, crown layer, axis and growth unit scale each consists of only one representative node type. In a straightforward manner, they are the stand type, tree type, crown type, crown layer type, axis type and growth unit (GU) type respectively. Leaf, bud, internode, root and marker types are the basic building organs of the modelled beech trees. The internode type acts in place of tree segments while the marker type is used as virtual and invisible markers in the topological structure for self-pruning. The root type is used for basal nodes for consolidation in the carbon transportation model. At the finest scale, the metabolic species types represent constituents of the Shikimate pathway each accounting for quantity. The node types at organ scale are inter-connected by branching and successor edges to allow these relationships between the instantiated nodes of these types. (Here we permit more than needed, since in reality, e.g., a leaf will not be succeeded by a bud. However, there is no necessity to include all possible restrictions in the type graph.) An identical inter-connection by branching and successor edge types is also specified for the types at the metabolic network scale. Aside from their role in the type graph, each type is declared as a module (in terms of the programming language XL) [91] with attribute values that contribute to simulation.

Simulation initialization begins with a procedure to create three stands

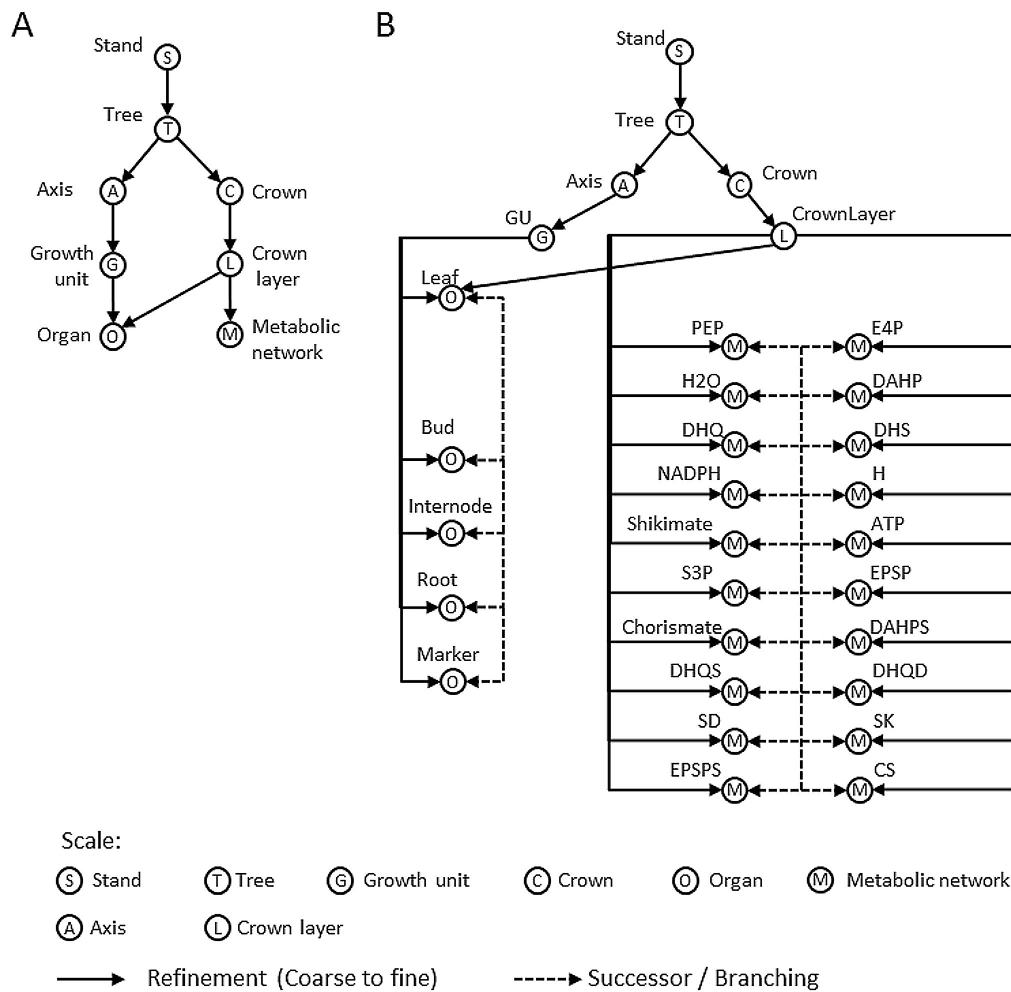


Figure 6.16: An illustration of the structure-of-scales (A) and type graph (B) for modelling three beech stands under ozone exposure.

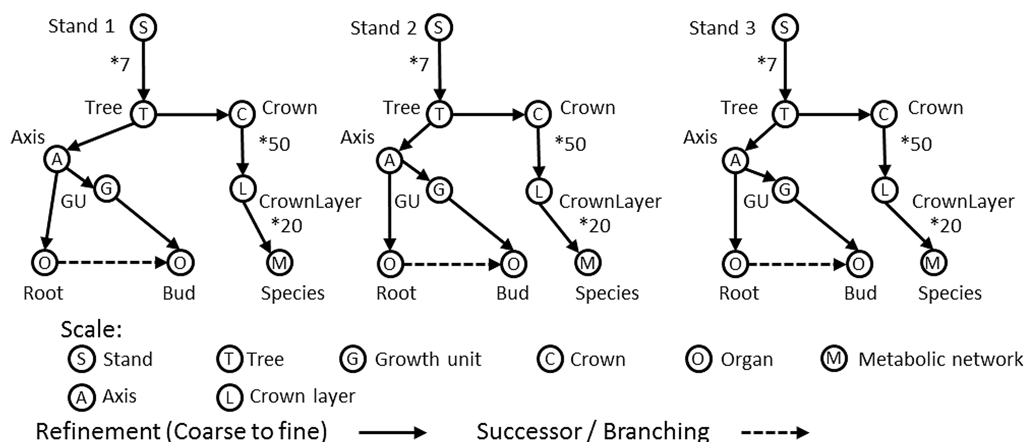


Figure 6.17: An illustration of the initial instanced graph for modelling three beech stands under ozone exposure. Edges labelled with an asterisk (*) and number represent multiple connections to the specified number of distinct nodes.

with ozone AOT40 (accumulated ozone exposure of 40 parts per billion) of 10000, 25000 and 40000 (μg m^{-3}).h. For each stand, the refinements to seven individual beech trees are created each with a single axis, a growth unit (GU), a bud and a root node. The crown decomposition of each tree is created using a crown node and a series of crown layer nodes. Every crown layer node is decomposed into the twenty species of the Shikimate pathway, connected in a specific order closely resembling the reaction sequence. The type graph is specified using the XL code in Listing 6.2.

```
^ /> TypeRoot /> Stand /> Tree
  [/> Crown /> cl:CrownLayer /> {# PEP E4P ... #} ]
  /> Axis /> GU />
  {# Bud Internode Root l:Leaf Marker #},
  cl /> l;
```

Listing 6.2: XL Type Graph Construction for beech stand

The last line `cl /> l` establishes the refinement of crown layers (`CrownLayer`, `cl`) to leaves (`Leaf`, `l`) in addition to the refinement from growth units (`GU`) to leaves (`Leaf`, `l`). Figure 6.17 depicts a condensed instanced graph of the model upon initialization.

Multiple steps simulate growth of the beech trees in the stands per year.

No.		Step Description
1		Light model, ray tracing and irradiance of leaves
2	*	Mean irradiance in crown layers
3	*	Photosynthetic depreciation in crown layers and metabolic network simulation
4	*	Photosynthesis and carbon assimilation
5		Transportation, allocation and distribution
6		Secondary growth
7		Primary growth (segment elongation)
8		Branch fall and bending
9		Update of refinement relationships between crown layers and leaves
10	*	Aggregation of data in tree and stand nodes

Table 6.2: Overview of the steps executed within one simulation step for modelling the three beech stands under ozone exposure. Steps with an asterisk (*) indicate usage of the multiscale approach.

Table 6.2 gives an overview of the steps in order. Of particular interest for our results are the application of (multiscale) grammar formalisms in step ten and steps two to four that endorse the multiscale framework. Figure 6.18 summarizes the problem categories, scale dependencies and scale interactions for steps two to four (see Section 4.1 on the *problem categorization*, *scale dependency*, and *scale integration* components of the multiscale modelling framework).

A photon tracing technique following the implementation by [76] is performed to obtain irradiance for single leaves.

The photosynthetic production for each leaf is multiplied by a normalized capacity factor obtained from the crown layer in which it resides. This inter-scale dependency is categorized as a multiscale problem requiring feedback from the macro-scale globally (i.e. scale feedback is required not only at localized domain) and illustrated as Framework A in Figure 6.18. The organ (leaf) scale takes the role of dependent scale and the crown scale is the independent scale. Scale integration begins with initialization (Framework A) by computing the mean irradiance of each crown layer using the individual irradiance of leaves within:

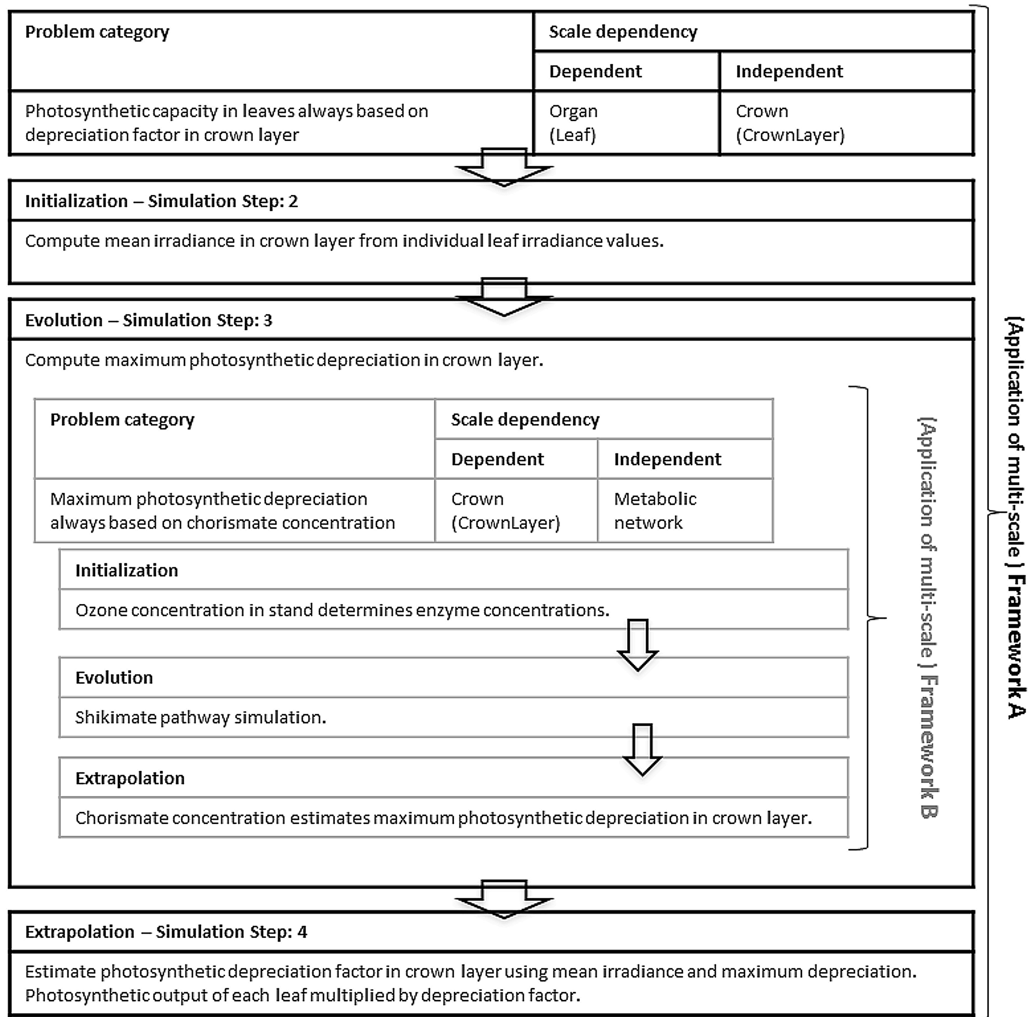


Figure 6.18: Schematic diagram for the application of the multiscale framework for determining photosynthetic capacity and production of leaves at their respective crown layers. The first application (Framework A) estimates photosynthetic production of each leaf using a capacity factor obtained from the crown layer. The nested application (Framework B) estimates the maximum photosynthetic depreciation at each crown layer using ozone concentrations.

```
layer.lightMean = mean((* layer Leaf *).lightIntercepted);
```

Notice that the query for Leaf follows layer without any edge specification due to the pre-defined refinement relationship in the type graph. This corresponds to step two of the simulation steps (see Table 6.2).

In evolution (Framework A), maximum photosynthetic depreciation in each crown layer is estimated from chorismate concentration. This dependency is again formulated as a multiscale problem requiring feedback from the metabolic network scale globally. It is an application of the multiscale framework nested within Framework A, illustrated by Framework B in Figure 6.18. The dependent scale is the crown scale while the independent scale is the metabolic network scale. In initialization (Framework B), ozone AOT40 is used to determine the concentrations of enzymes raised by hypersensitivity to ozone. For example, the concentration for DAHPS is specified by the rule-based statement as shown in Listing 6.3.

```
s:Stand (/>)* CrownLayer
[dahps:DAHPS][dhqd:DHQD][sd:SD][epsps:EPSPS] ::> {
    dahps.con=DAHPS.conMin +
              (s.ozone * DAHPS.conRange);
    ...//other 3 enzymes
}
```

Listing 6.3: Specifying enzyme concentrations using ozone conditions in tree stand

No specific edge specification between `CrownLayer` and the metabolic species is required due to the pre-defined refinement relationships in the type graph. The rest of the enzymes have their concentration values reset. In evolution (Framework B), the metabolic reactions are simulated. First, the rates of all reactions are determined. The concentrations are then updated using the computed rates. In extrapolation (Framework B), chorismate concentration is expressed as a percentage of a maximum chorismate concentration. This percentage is interpreted as the maximum photosynthetic depreciation for the crown layer. Returning to extrapolation (Framework A), mean irradiance and maximum photosynthetic depreciation estimate the normalized capacity factor in the crown layer. The photosynthetic production of each leaf is reduced with a multiplication by the capacity factor of the crown layer it resides in. An example of the code is shown in Listing 6.4.

```
cl:CrownLayer lf:Leaf ::> {
    lf.carbonAssimilated = calculatePS(lf);
    lf.carbonAssimilated *= cl.photosynCapacity;
}
```

Listing 6.4: Reducing photosynthetic production of leaves based on crown layer capacity

where `calculatePS` computes the photosynthetic output from leaf `lf` and the operator `*=` multiplies the output by the normalized capacity factor (`photosynCapacity`) from the crown layer.

Carbon assimilation, transport and allocation are implemented at the organ scale with the topological structure of individual trees using the methods by [85] and [165]. Allocated carbon is used for secondary growth, i.e. incrementing the diameter attribute values of Internode nodes in the graph.

The number of primordial leaves and consequently, the number of internodes is computed for each bud based on the diameter of the nearest internode [33]. Based on empirical measurements by Schober [157], the length of internodes based on the age of the tree is determined. Primary growth demonstrates the application of multiscale grammar rules, establishing relationships from the tree to the organ scale in a concise statement. For example, the code in Listing 6.5 creates a number of internodes, lateral buds and leaves specified by `numInternode`. Figure 6.19 illustrates an iteration of this sample code for the production of two internodes from a bud.

```
t:Tree a:Axis g:GU Bud ==> t a g
                          for(1:numInternode)
                          (Internode [Axis GU Bud])
                          GU Bud;
```

Listing 6.5: Beech growth with multiple scales

After a simplified simulation of branch fall and bending, the refinement from crown layers to leaves is updated based on the vertical height of leaves. Finally, data is aggregated from organ to tree scale and from tree to stand scale. Ozone AOT40 values in the three stands are updated using trends proposed by [41]. Figure 6.20 shows the stands after fifteen simulation years.

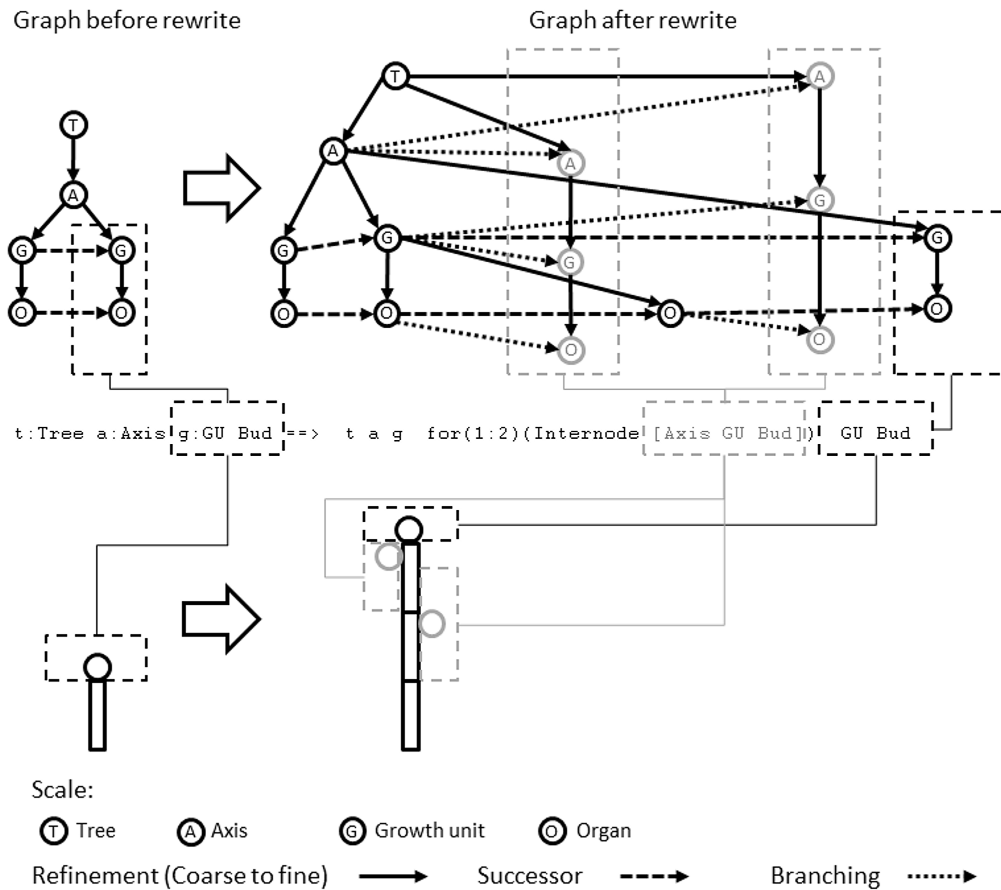


Figure 6.19: Example of an execution of a multiscale rule in XL. The instanced graph is shown on top and the rule in XL code is shown in the middle. The bottom shows a corresponding geometrical representation of the instanced graph. In this rule, a bud is queried on the left-hand side of the rule along with the chain of encoarsements up to the tree scale. On the right-hand side of the rule, light grey graph nodes (with light grey XL syntax and geometrical picture) represent the two lateral buds produced, along with their chain of encoarsements up to the axis scale.

Figure 6.20: Screenshot of the three beech stands after sixteen simulation years. The left stand is exposed to the least ozone, the middle one to moderate ozone and the right one to the most ozone.

6.6 Stand Dynamics and Morphological Developments of Conifers

This examples describes the development of a multiscale model that consists of stand dynamics, crown profile development, height growth, diameter at breast height (dbh) growth, and branching structure development of individual trees. Emphasis is placed on the rule-based (graph) data structure-oriented programming approach, and the integration of quantitative values at different scales for a consistent multiscale model. For this example, empirical models are adopted from literature and combined. The rest of this section is divided into sub-sections describing the multiscale graph structure, model initialization, the germination model, the growth model, and the mortality model.

6.6.1 Multiscale Graph Structure and Model Initialization

An overview of the scales and entity types required in the model is first constructed using the structure-of-scales and type graph (Figure 6.21).

Three scales make up the model. The Stand scale is the coarsest scale and comprises the entity types *Stand*, which represents a collective object for a forest stand, and *Grid*, which represents a unit of the regularly divided stand area. The Tree scale comprises an entity type *Tree* that represents a single fir tree. The finest scale Organ has entity types *Bend*, *I*, *Bud*, and *Trunk*. *Bud* represents a bud, *I* represents an internode, *Trunk* represents

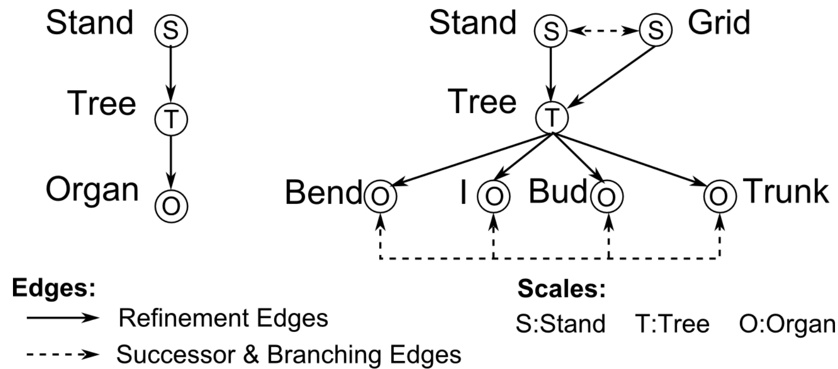


Figure 6.21: Graph construct for model simulating stand dynamics and developments of crown profile, height, dbh, and branching structure of virtual fir trees. Left: structure-of-scales. Right: type graph.

an internode along the trunk or main stem, and *Bend* represents a connection between a lateral branch and the trunk. The entity types, otherwise known as modules in the programming language XL, are declared in code. In the initial procedure `init()`, a forest stand with an evenly divided grid floor is created using the code shown in Listing 6.6.

```
{grids = createGridFloor();}
//grids is a double array of Grid objects
Axiom ==> Stand
          for(int i:(1: S_GRID_COUNT_X))(
            for(int j:(1: S_GRID_COUNT_Y))(
              [grids[i][j]]
            )
          );
```

Listing 6.6: Creation of grid floor for forest stand

The procedure `createGridFloor()` initializes a double array (rows and columns) of Grid nodes. To initialize the instance graph, a rule with query for `Axiom` creates a Stand node that is connected to each Grid node via a branching edge. The `init()` procedure additionally contains the rule for creating the type graph (see Listing 6.7), as illustrated in Figure 6.21, and connecting it to the root node ^.

```
==>> ^ {# Stand Grid #} /> Tree />
```

```
      {# Bend I Trunk Bud #};
```

Listing 6.7: Creation of type graph for forest stand

The symbols `{#` and `#}` enclose nodes in a clique, connecting each pair within bi-directionally by successor and branching edges.

Upon completion of `init()`, a procedure `run()` is invoked once for every simulation year. Three major simulation sub-steps, described by the next three sub-sections, are included in `run()`.

6.6.2 Germination

A constant pool of germinating seeds is assumed to reside in the virtual forest stand each year. The procedure for the first sub-step of `run()`, `germinate()`, contains the rule shown in Listing 6.8 that appends graph nodes representing seedlings to the instance graph.

```
s:Stand ==> s for(int i:(1:S_SEED_POOL))(
                 {x = random(0,S_DIM_X);
                  y = random(0,S_DIM_Y);}
                 [createTree(x,y) createTrunk Bud(0)]
               );
```

Listing 6.8: Germination rule

In the second and third lines of the rule, a Java code block (between curly brackets) is embedded to generate random 2-d coordinates for the position of a new tree. `createTree(x,y)` and `createTrunk` are procedures that return new graph nodes with entity types $Tree$ and $Trunk$ respectively. All in all, a total of `S_SEED_POOL` nodes with entity type $Tree$ are newly refined from each $Stand$ node. Each new $Tree$ node is refined to a new $Trunk$ node succeeded by a new Bud node.

Internally, the procedure `createTree(x,y)` establishes a refinement edge from a $Grid$ node to the new $Tree$ node it creates based on the position given by `x` and `y` (see Listing 6.9).

```
{Tree t = new
   Tree(x,y,T_INIT_HT,T_INIT_CI,T_INIT_DBH,
                     T_INIT_AGE,T_CR,treeCwmax(0),
                     T_INIT_HCBASE,T_INIT_HD,
                     T_INIT_NUMINT,T_CROWN_SHADE,
```

```
                          T_RPMAX);}
==>> grids[x/S_DIM_X][y/S_DIM_Y] t <+ ^;
```

Listing 6.9: Grid to tree refinement

The *Grid* node representing the grid unit in which the tree resides is identified from the double array using a simple division of the coordinates by the grid dimensions, `S_DIM_X` and `S_DIM_Y`. The graph root node `^` is connected to the new *Tree* node via a branching edge for visualization purposes, since our tool GroIMP currently shows in its 3D view only nodes accessible from the root by a path composed exclusively of successor and branching edges. Parameters for a new *Tree* node are mostly constants except for maximum crown width, which is computed by the procedure `treeCwmax`. Empirical data for maximum crown width of fir trees was extracted from [157]. The data was fitted and computed as

$$cw = 8.331566/(1 + 5.011992 * age^{1.011805})^2 + 0.194259$$

where cw is the maximum crown width for a tree with age age. Similar to `createTree`, `createTrunk` consists of a rule to connect the graph's root node to new *Trunk* nodes.

6.6.3 Growth

The second sub-step of `run()`, `grow()`, contains procedures and rules to compute competition among trees, growth of individual trees, and development of organs and finer structures for each tree.

6.6.3.1 Stand Competition Index

An empirical model of forest stand development based on [21] is utilized. Dbh of trees in each grid unit is reset and summed using the rules in Listing 6.10.

```
g:Grid ::> {g.dbhSum = 0; g.dbhSumInRange = 0;}
g:Grid t:Tree ::> {g.dbhSum += t.dbh;}
```

Listing 6.10: Sum of dbh for competition index computation

The first rule resets the value of parameters `dbhSum` and `dbhSumInRange` for each *Grid* node. The second rule aggregates the dbh of each tree, `t.dbh`, residing in each grid unit into `g.dbhSum`.

The effects of competition between trees occur within a specified distance range `S_COMPETE_RANGE`, which is resolved into a number of grid units depending on the dimensions of a grid unit. The sum of dbh in grid units within the range `S_COMPETE_RANGE`, `dbhSumInRange`, is aggregated for each *Grid* node using conventional Java iterations on the double array `grids`. Correction is performed for grid units along the marginal areas of the virtual stand by compensating for out-of-range areas using the average `dbhSum` of within-range grid units.

With the accumulation of dbh in grid units, competition indices of trees are set using the rule in Listing 6.11.

```
g:Grid t:Tree ::> {t.ci = g.dbhSumInRange/t.dbh;}
```

Listing 6.11: Setting competition index for trees

where `t.ci` is the competition index for tree `t` residing in the grid unit represented by *Grid* node `g`.

6.6.3.2 Individual Tree Growth

Empirical data for height and dbh of fir trees is extracted from [157]. The data was fitted and computation of height and dbh are as follows:

$$ht = 0.023777/(0.000559 + age^{-1.896382}) + 0.400637$$

$$dbh = 0.225899 * age^{1.142437+(-0.000378*age)} + 1.012812$$

where ht, dbh, and age are the height, dbh, and age of a fir tree respectively. Given height, dbh, and age, crown ratio (vertical proportion of height that the crown occupies) is computed based on the model by [45]:

$$cr = 1 - e^{-(0.55243+5.026/age)*dbh/ht}$$

where *cr* is the crown ratio of the tree.

A rule (shown in Listing 6.12) in `grow()` utilizes the empirical fittings and pre-computed competition indices to determine the development of each tree.

```
t:Tree ::> {
    //poten. ht inc.
```

```
float htPotDelta = treeHt(t.age + 1) - t.ht;

//ht inc. w. competition
t.hd = cDelta(htPotDelta, t.ci, t.cr);
t.ht += t.hd;

//poten. dbh inc.
float dbhPotDelta = treeDbh(t.age + 1) - t.dbh;

//dbh inc. w. competition
t.dbh += cDelta(dbhPotDelta, t.ci, t.cr);.
t.age ++; //tree age

//crown ratio
t.cr = crownRatio(t.dbh,t.ht,t.age);

//maximum crown width
t.cwmax = treeCwmax(t.age);

//height to base of crown
t.hcbase = (1-t.cr) * t.ht;
}
```

Listing 6.12: Multiscale tree development based on competition indices

The procedures `treeHt`, `treeDbh`, and `crownRatio` contain the aforementioned empirically fitted formulas for height, dbh, and crown ratio of fir trees. Given the succeeding age of a tree (`t.age + 1`) , the potential height increment, `htPotDelta`, and potential dbh increment, `dbhPotDelta`, are computed. These potential increments are provided as inputs to the competition model (based on [21]) specified in the procedure `cDelta` to obtain actual increments in the competitive environment. The competition model is computed as follows:

$$d = dpot * (0.26325 + (2.11119 * cr^{0.56188} * e^{-0.26375*ci-1.03076*cr}))$$

where d is the actual increment, $dpot$ is the potential increment, cr is the crown ratio, and ci is the competition index of the tree. Lastly, age, crown ratio (cr), and maximum crown width ($cwmax$) are updated for each tree.

6.6.3.3 Structural and Architectural Development

In this section, we first describe apical growth of a tree's trunk, followed by elongation of lateral first order branches. Bending and senescence of lateral branches are described at the end.

Development of tree trunks is specified by the rule in Listing 6.13.

```
t:Tree b:Bud, (b.order == 0) ==> { int numInt =
   (t.hd / I_ELONG0);}
        t for(int i: (1:numInt)) (
            Trunk(I_ELONG0, I_INIT_DBH) RH(I_PHYLLO)
            [Bend(I_ANGLE, 0) I(I_ELONG1) Bud(1)] )
        Trunk(t.hd%I_ELONG0, I_INIT_DBH)
           RH(I_PHYLLO) Bud(0);
```

Listing 6.13: Tree trunk development

where `numInt` is the number of internodes to be created along the trunk and `t.hd` is the precomputed actual height increment for the tree. The for-loop inserts *Trunk* nodes according to a phyllotaxy constant `I_PHYLLO`, each with a lateral branching angle `Bend(I_ANGLE, 0)`, internode `I(I_ELONG1)`, and bud `Bud(1)`. The final apical internode is represented by a *Trunk* node with length `t.hd % I_ELONG0`, remainder of the division of actual height increment by internodal length. This rule operates only for the bud with order zero, i.e. apical bud for the main stem or trunk, as indicated by the query condition `(b.order == 0)`.

The development of lateral first order branches is specified by the rule in Listing 6.14.

```
t:Tree b:Bud, (b.order == 1) ==> { Point3d locB =
   location(b);
        float dist =
           distToTrunk(t.x,t.y,locB.x,locB.y);
        if(locBud.z > t.hcbase) {
            float rp = 1-((locBud.z -
               t.hcbase)/(t.ht - t.hcbase));
            float cwah = (cwah(t.cwmax, rp, t.ht,
               t.dbh)/2);
        }
        float elong = cwah - dist; if(elong < 0)
           elong = 0;}
```

```
        t if(elong > 0)(RU(I_TROPISM) I(elong)
          Bud(1))
          else(b);
```

Listing 6.14: First order branch development

The 3D position, `locB`, of the first order bud is first computed. With its position, the minimum (perpendicular) distance, `dist`, of the bud from the trunk is computed. If the bud is not below `t.hcbase`, the height to the base of the crown, we compute the vertical relative position, `rp`, of the bud in the crown. `rp` is provided as input to a crown profile model based on [73] to compute `cwah`, the crown width at a specific height as follows:

$$f = 0.929973 - 0.135212 * rp^{0.5} - 0.131316 * (ht/dbh)$$

$$cwah = cwmax * rp^{f}$$

where f is a coefficient computed using the vertical relative position in crown, height, and dbh of the tree. *cwmax* is the pre-computed maximum crown width of the tree. Elongation, `elong`, of the lateral branch is computed as the difference between `cwah` and `dist`. A rotational node `RU(I_TROPISM)` with tropism angle `I_TROPISM`, and an I node representing an internode with length `elong` are appended to the graph if *elong* is positive. Figure 6.22 illustrates the various parameters associated with the crown profile graphically.

Mechanical bending of the branches is simulated by modifying the angles of *Bend* nodes. Branches originating below the pre-computed minimum height to crown base are removed. These two operations are specified with the rules in Listing 6.15.

```
t:Tree b:Bend ::> {if(location(b).z > t.hcbase)
   b.angle+=I_BEND;
        else cutBranch(b);}

b (-->)+ Node ==>> ;
```

Listing 6.15: Mechanical bending

The first rule checks if the height `location(b).z` of the *Bend* node `b` is higher than the minimum height to crown base `t.hcbase` of the tree. If so, the angle of the *Bend* node `b` is incremented by `I_BEND` degrees. If not, the procedure `cutBranch` is invoked with `b` as input parameter to remove

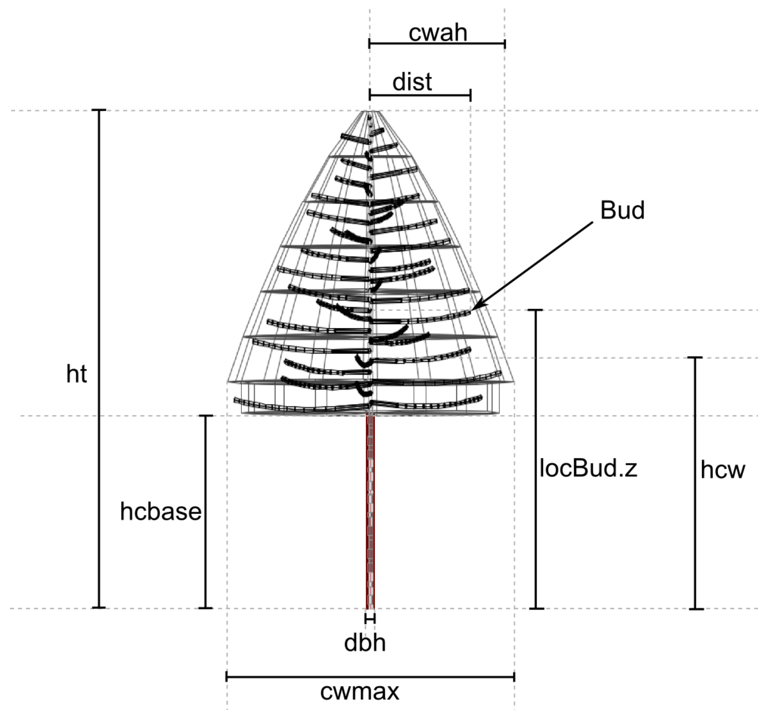


Figure 6.22: Illustration of variables - height (ht), diameter at breast height (dbh), maximum crown width (cwmax), bud's minimum distance to trunk (dist), crown width at height (cwah) for vertical height (hcw), height to base of crown (hcbase), and bud's vertical position (locBud.z). Relative position in crown (rp) is 1 − ((locBud.z − hcbase) / (ht − hcbase)).

nodes representing the branch. The second rule is specified in the `cutBranch` procedure. It queries for the node `b` as well as all nodes `Node` with a path from `b` and replaces them with an empty production graph, effectively deleting these nodes from the instance graph.

6.6.4 Mortality

The third and last sub-step of `run()`, `mortality()`, contains procedures and rules that simulate the death of trees in the virtual forest stand. Mortality is based on the model by [21] and is specified by the rule in Listing 6.16.

```
t:Tree /> n:Node ==> {float prob = S_LIVE_JUVENILE;
         if (t.age >= S_AGE_COMPETE) prob =
            probLive(t.cr, t.ci);}
         if (probability(prob)) (t /> n);
```

Listing 6.16: Mortality

A $Tree$ node and all nodes with a refinement edge connection from it are queried from the instance graph. If the queried tree is younger than `S_AGE_COMPETE`, its probability of survival, `prob`, is `S_LIVE_JUVENILE`. If it is of age `S_AGE_COMPETE` or older, its probability of survival is computed from its crown ratio (`t.cr`) and competition index (`t.ci`) in the procedure `probLive` as

$$k = -0.0023 * ci^{0.65206}$$

$$p = 1.02797 * cr^{0.0379} * e^{k}$$

where k is a cofficient computed using the competition index ci, and p is the probability of survival, computed from the tree's crown ratio cr and k. The tree's survivial is determined by the procedure `probability` with `prob` as input parameter. If the tree survives, the nodes are specified as they were in the production statement on the right-hand side, leaving them intact in the instance graph. Otherwise, the production statement is left empty, effectively removing the nodes representing a tree from the instance graph.

A list of parameter values (capitals in code statements) is provided in Table 6.3. Figure 6.23 shows screenshots of the model.

Remark: The contents in this chapter are published in [128], [130], and [129].

S_AGE_COMPETE	8	T_INIT_HCBASE	0
S_DIM_X	16	T_INIT_HD	0
S_DIM_Y	16	T_INIT_HT	0.1
S_GRID_COUNT_X	4	T_INIT_NUMINT	0
S_GRID_COUNT_Y	4	T_RPMAX	0.1
S_LIVE_JUVENILE	0.8	I_ANGLE	80
S_SEED_POOL	26	I_BEND	1
T_CR	0.3	I_ELONG0	0.2
T_CROWN_SHADE	0.9	I_ELONG1	0.1
T_INIT_AGE	0	I_INIT_DBH	0.0575
T_INIT_CI	0	I_PHYLLO	132
T_INIT_DBH	1	I_TROPISM	-1.3

Table 6.3: Constant parameter values. Prefix S, T, and I, for stand, tree, and internode (organ) scales respectively.

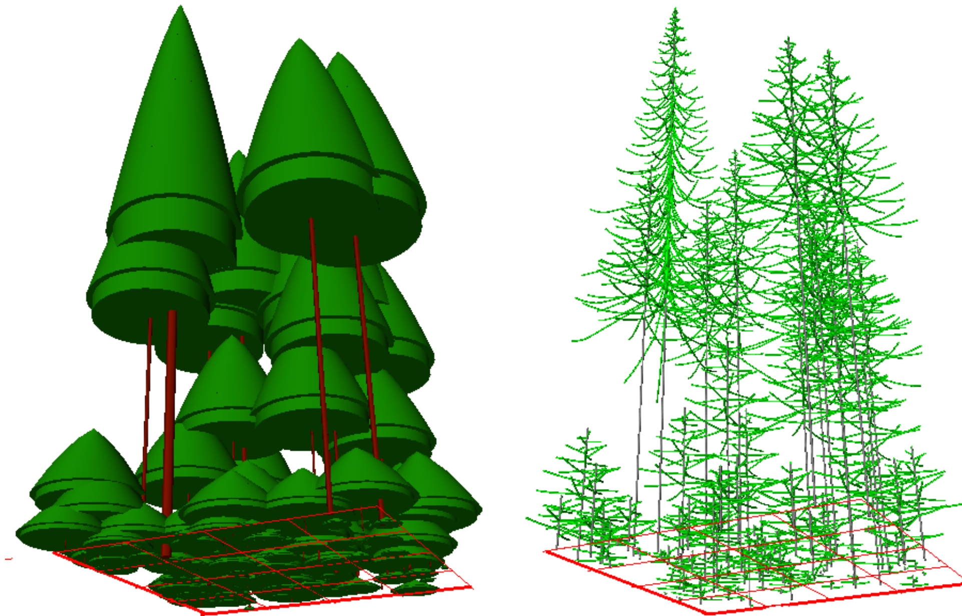


Figure 6.23: Illustration of 154 trees in a 265 year old stand. Left: Visualization of the crown profiles. Right: Visualization of organ scale with internodes.

Chapter 7

Conclusion

This chapter concludes the thesis by addressing the research questions and objectives raised in Sections 1.4.1 and 1.4.2. Several concluding remarks based collectively on the examples in Chapter 6 are also made together.

7.1 Answers to Research Questions

How can 3-dimensional structures be described consistently at several spatial scales at once, based on the graph representation used in XL?

Chapter 4 defines a three-part graph data structure that contains a structure-of-scales (Section 4.3.1), a type graph (Section 4.3.2), and an instanced graph (Section 4.3.3) suitable for describing 3-dimensional structures at several spatial scales.

The notion of plant modularities and MTG topology is shown in the structure-of-scales containing plant, axis, growth units and organs in the example illustrated in Section 6.5 (beech tree stands under ozone exposure). In that example, two forms of static spatial division are illustrated. A continuous concept of three-dimensional space is used for the random positioning of individual beech trees in a stand area. MTG modularities and stand space are additionally merged with a discrete spatial division employed by the crown layers. Dynamic spatial segregation is shown in the example in Section 6.4 that has an evolving tree crown space. Network-based systems such as the metabolic network in the beech trees under ozone exposure are successfully embedded in the graph data structure.

A single graph data structure is used to represent both geometric spaces and topological structures in the two above mentioned examples. This allows the use of rules to access and modify relationships between the spaces, e.g. relating a leaf with a canopy layer. Moreover, in this manner, topological changes directly impose a change in the graph data structure, reducing the need to synchronize changes to the geometric space. For example, the removal of a graph node (e.g. simulating the death of a leaf) imposes the removal of all edges connected with it, including the edge from the graph node representing the canopy layer containing the leaf. If the two spaces are maintained in separate data structures, an explicit procedure to remove the reference to the leaf from the geometric space (canopy layer) is required.

How can multi-scaled graphs be transformed by rules?

Section 4.4 in Chapter 4 defines transformation techniques for transforming multi-scaled graphs. The techniques rely on the three-part graph data structure and are defined as extensions to the SPO with operators (Section 3.3.2.3) formalism in [91]. In addition, a framework inspired by multiscale modelling in other domains of science is introduced in Section 4.1. The framework puts into perspective the combination of rule-based modelling and classical multiscale modelling techniques [46].

Refering to the examples in Section 6.4 (crown generation) and 6.5 (beech trees under ozone exposure), a comparison of scale dependencies and extrapolation methods in the multiscale modelling framework are made in [130].

What are the various aspects of consistency in multi-scaled graphs? How can consistency checks on these aspects be performed efficiently on multi-scaled graphs?

This question has not been answered and is left for future work.

How can transformation, query and rendering of multi-scaled graphs be performed efficiently?

Two considerations of efficiency have been made:

- The large number of queries into the type graph for multiscale rules is recognized. To reduce the number of traversals of the type graph (for every match of a query), a cache mechanism is introduced as described in Section 4.6.3.
- The search order for queries may need to be optimized. This consideration is however not concretized and is left for future work.

How can the programming language XL be extended for multi-scaled graphs?

The syntax of XL is modified as described in Section 4.5 to support multi-scaled graphs. In addition, the use of the observer design pattern is introduced as a high level programming approach to complement multiscale modelling.

How can classical, and, possibly new Level-of-Detail methods be incorporated in the multi-scaled graph approach of XL?

Chapter 5 introduces an incremental level-of-detail computation method for branching structures. The chapter also includes a multiscale graph traversal method catered for interpretation of the multiscale graph data structure as a scene graph.

What are the various test data and scenarios that can considerably challenge the performance, rendering quality and integrity of multi-scaled structures?

Several examples and demonstrative models are presented in Chapter 5 and 6. These examples and models ([128], [130], [129], [127]) pose challenges in terms of performance, rendering quality and integrity of multi-scaled structures.

Are there any unique or isolated scenarios that result in unsatisfactory performance, rendering quality or integrity of multi-scaled structures? What are the differences in performance and rendering quality of multi-scaled structures as compared to existing or alternative implementations?

Multiscale rules in general result in shorter implementation code as compared to single-scale implementations. These advantages currently come with a price of slower executions in comparison to single-scale rules. This occurs at least for the existing implementation in GroIMP, which does not utilize possible speed-up techniques for graph matching. The search order for queries may potentially improve in this aspect and is left for future work. In addition, edge discrimination in queries where edge connections to each graph node are grouped according to their labels may potentially speed up query for multiscale rules.

What workflows and interactive Graphical User Interface (GUI) elements are suitable for users to control the transformation, query and rendering of multi-scaled structures?

In addition to the new syntax and features of the XL language, selection of visible scales is made possible by a pop-up menu in the 3D panel of GroIMP. The items of the pop-up menu are listed according to the scales in the structure-of-scales constructed in the graph data structure. Each item or scale is a checkbox in the menu and is visibled if checked.

7.2 Concluding Remarks

The work in this thesis has introduced advances in graph rewriting catered primarily for functional-structural plant modelling. The theory is implemented in the XL programming language and the open-source software GroIMP. The nature of the graph structures produced are intepreted as geometry, topology and arbitrary data structures. Advances in XL must therefore involve concurrent considerations from these different perspectives. Much of the considerations made in this thesis are centered on topology, particularly on the unique refinement relationship, leaving much to be desired in the other aspects. Future work in the aspects of high performance graph processing (the data structure aspect) and real-time rendering (the graphical and geometrical aspects) can serve as continued work from this thesis.

Chapter 8

Appendix: Interfaces with Other Software

8.1 The MTG File Interface

The *multiscale tree graph* (MTG) is a multiscale data structure defined mathematically by Godin and Caraglio [67]. Accompanying this data structure is a file format that allows text encoding for storing and exchanging MTG data. The file format is documented online [1]. GroIMP is extended with an interface to import and export MTG files. This section documents the implementation of the interface as well as basic usage instructions.

An MTG can be represented using the *instanced graph* data structure introduced in Section 4.3.3. Hence, no conversion or translation of graph topology is necessary during import and export. However, geometrical interpretation of an MTG is different from an instanced graph because MTG nodes typically contain absolute (not relative) cartesian coordinates that position them in 3-dimensional space. The orientation of a 3D object represented by an MTG node is then computed as the vector resulting from the difference between the position of the node and the next successor or branching node. MTG nodes rely on a set of permanently named attributes for geometrical interpretation. For example, the *length* attribute of an MTG node must be named "Length". The complete list of permanent MTG attribute names can be found in the online documentation. It is noteworthy that more than one interpretation of the MTG file format exist ([68], [136], [14]), giving room for minor differences such as the necessity for space characters or empty lines in

the file content. Consequently, although the MTG interface implemented in GroIMP follows the online documentation (the above mentioned link), minor adjustments to file contents may be necessary for successful import to or reading of MTG exports from GroIMP.

An import of an MTG file in GroIMP is organized into two general steps. The first step involves the reading of the header sections of the MTG file. The header sections are the "CODE", "CLASSES", "DESCRIPTION", and "FEATURES" sections in this order starting at the beginning of the file. These sections do not contain the actual graph nodes, edges, and attribute values but meta information such as the version, classes, topological constraints, and node attributes. These can be represented in our multiscale graph data structure (see Section 4.3) and XL using *modules*, edges' connections in the *type graph* (although no type graph is generated by the import), and *module attributes* respectively. At the end of the first import step, GroIMP creates and compiles an XL script containing *module* declarations (prefixed with "mtg_" followed by the MTG class name) equivalent to that specified in the MTG headers. The second import step proceeds to read the body of the MTG file, which contains the graph nodes, edges, and attribute values. GroIMP constructs the graph following the topology specified in the file by creating node instances from the compiled *modules* and loading them with attribute values read from file. Each line the in MTG file body encodes one or more graph nodes that share a set of attribute values.

A so-called *dressing file* may accompany an MTG file. A dressing file provides additional data for visualization such as references to 3D meshes (e.g. .smb files), texture images, etc. The interface developed in GroIMP does not yet support loading of dressing files. A default set of parameters identical to that used in the AMAPmod software [68] is used to compute geometrical information for visualization in GroIMP.

Once imported, an MTG in GroIMP is rooted by an additional root node of the unique class `MTGRoot`. This node contains all meta information loaded from the MTG headers. In addition, it is interpreted geometrically as a 3D mesh by GroIMP for rendering. To render the MTG, the method `plantFrame(scale,distance)` needs to be invoked on the `MTGRoot` node, where `scale` is the scale to visualize and `distance` is the 3D distance per unit length in the MTG. The implementation of the `plantFrame` method is based on the AMAPmod software [68] for MTG geometry computation.

A class diagram of the classes in the interface package "de.grogra.mtg" is shown in Figure 8.1. The "MTGFilter" class provides the entry point for

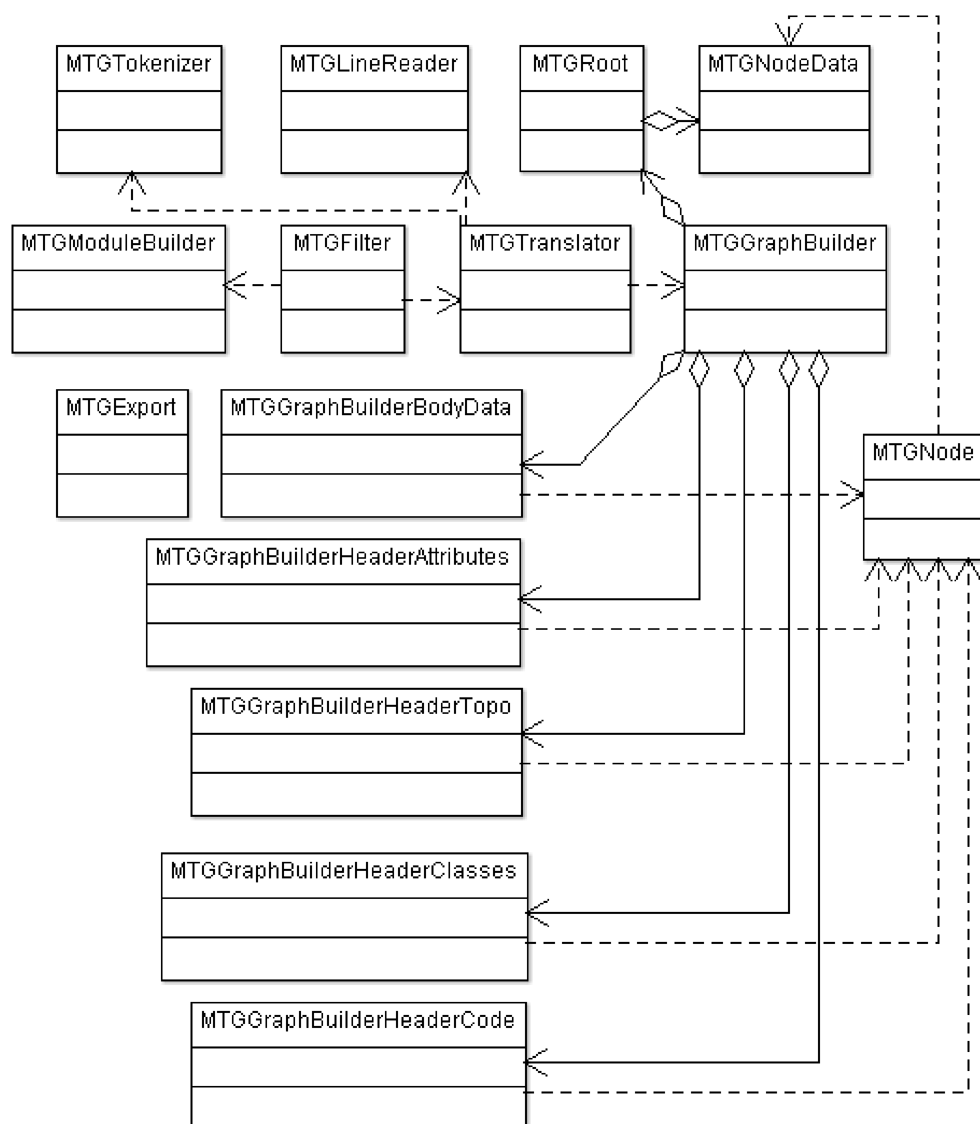


Figure 8.1: A condensed class diagram showing the classes implementing the import and export of MTG files in GroIMP. Some peripheral or subclasses are not shown to avoid cluttering the diagram.

MTG file import. It dedicates the interpretation of MTG file content to the "MTGTranslator" class, which reads lines and tokenizes the contents using the "MTGTokenizer" and "MTGLineReader" classes. The "MTGTranslator" class relies on the "MTGGraphBuilder" class and its subclasses for constructing nodes and edges of an instanced graph representing the imported MTG. The subclasses of the "MTGGraphBuilder" class are named based on the sections of an MTG file. Each *module* created and compiled for an MTG type is an extension of the "MTGNode" class that contains a reference to the "MTGNodeData" class for storing attribute values. Standard MTG attributes such as length are not included in the "MTGNodeData" class, they can be directly retrieved from an "MTGNode" instance. Export of an MTG from GroIMP's graph is supported using the "MTGExport" class. As MTG files allow only single letter naming of types, e.g. "A" or "I", the export functionality trims the modules names in XL and retains only the first letter. If a letter is already used, GroIMP assigns a new letter. If all letters are used, an export error occurs. The export functionality expects the graph in GroIMP to be an MTG, i.e. without cycles and circles, throwing an exception if the graph is not MTG compatible.

Import of an MTG file is possible by directly opening an MTG file from the main menu in GroIMP. Export is executed by selecting the "View->Export" option in the 3D panel menu. A screen shot of a visualized MTG structure [136] imported into GroIMP is shown in Figure 8.2.

8.2 The Xplo and ArchiTree Interface

Xplo is a part of the AMAPstudio software [70, 2] for exploring, building, editing, visualizing, and filtering multiscale plant architecture data. It provides a platform for developing software plugins (called *modules* in the terminology used by Xplo developers) that model and simulate virtual plants and environments.

Xplo handles plant architecture data with the *ArchiTree* data structure, which is an improved Java implementation of the MTG formalism. It is an MTG capable of representing both topology and geometry. An illustration of an *ArchiTree* in comparison with a *multiscale instanced graph* (Section 4.3.3) is shown in Figure 8.3.

There are two significant differences between the two data structures. Firstly, in an *ArchiTree*, refinements (decompositions) of a node are not all

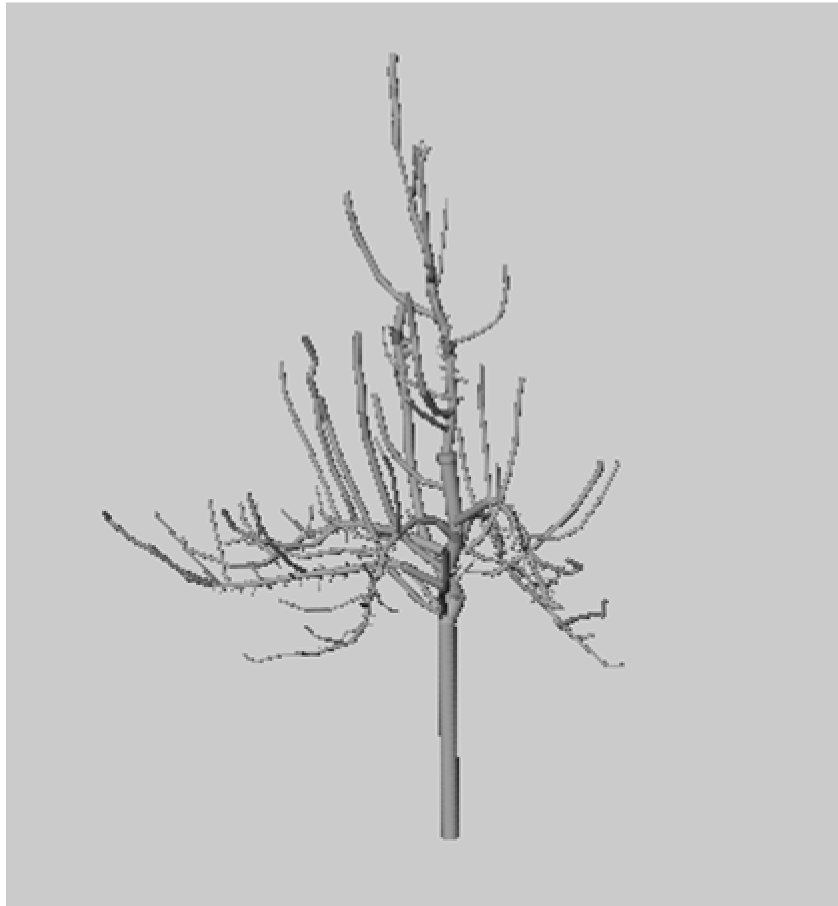


Figure 8.2: Visualization of imported MTG in GroIMP. The mesh is computed using the `plantFrame` method based on the AMAPmod [68] software.

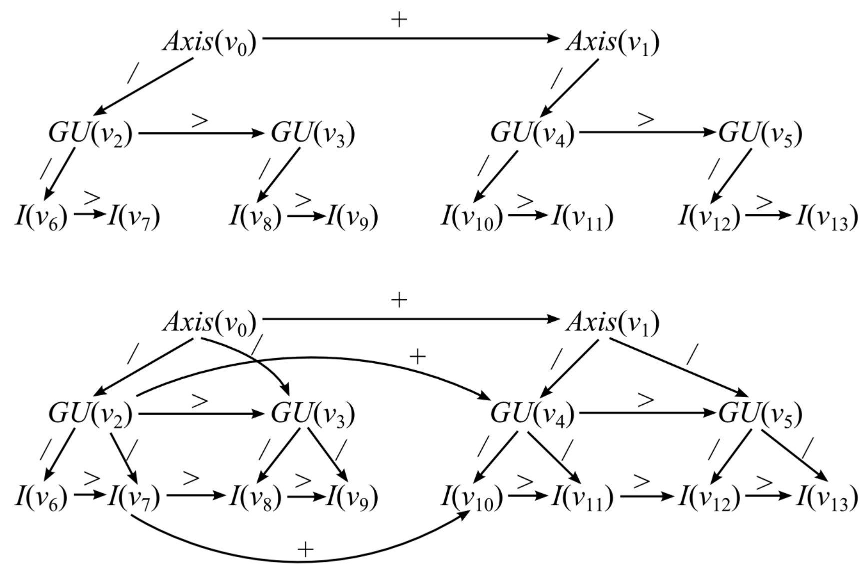


Figure 8.3: ArchiTree and multiscale instanced graph. The two connected graphs represent the same plant architecture with three scales - *Axis* (axes), *GU* (growth units), and *I* (internodes). Top: ArchiTree structure. Bottom: Multiscale instanced graph.

connected from their encoarsement. For example, in Figure 8.3, the (growth unit) nodes v_2 and v_3 are both refinements of the (axis) node v_0. In the *ArchiTree* representation, only the first node in a succession of nodes is connected from the encoarsement, hence only v_2 is connected from v_0. In the *multiscale instanced graph*, all refinement nodes are connected from their encoarsement, hence v_2 and v_3 are both connected from v_0. Secondly, a successor or branching (ramification) edge connection is represented only at one scale in the *ArchiTree* data structure. For example, in Figure 8.3, the (internode) node v_8 actually succeeds v_7 but no successor edge is established between them. This is because the same successor relationship is represented at the coarser scale from the (growth unit) node v_2 to v_3. In other words, successor and branching relationships are not duplicated across scales in an *ArchiTree*. On the other hand, the *multiscale instanced graph* represents these relationships at all scales, e.g. the edges $(v_7, >, v_8)$ and $(v_2, >, v_3)$ represent the same successor relationship at two different scales. One reason that the *ArchiTree* is capable of using less edges for the same plant architecture is that each *ArchiNode* (a node in an *ArchiTree*) contains a 4x4 affine transformation matrix that provides the absolute 3D position and orientation of the geometry represented by the node. In contrast, nodes in *multiscale instanced graphs* may contain only relative transformation information such as rotation angles.

A preliminary interface between GroIMP and Xplo is established. Source code version 3434 of GroIMP is built as a Java library file named GroImp.jar. A module is developed in Xplo to allow the running of an XL script in Xplo using this library version of GroIMP. The module class diagram is shown in Figure 8.4. `GIRelay` is a class for the graphical user interface (GUI) that allows the module to get initial parameters such as the folder location of the XL script, etc. `GIModel` contains the main methods for simulation using the XL script. `GIScene` represents a scene timeline and `GISketchLinker` is used to establish connections to the GUI panels. Two important methods in the `GIModel` class are `initializeModel` and `processEvolution`. Intuitively, these methods are used to initialize the plant model (analogous to the `init()` method in XL scripts) and to make one simulation step of the model (analogous to the `run()` method in XL scripts) respectively.

In this preliminary interface implementation, the methods `init()` and `run()` are mandatory for an XL script to run in Xplo. In the `GIModel` class, each call to `processEvolution` triggers a conversion of the *multiscale instanced graph* in GroIMP derived after invoking the `run()` method to an

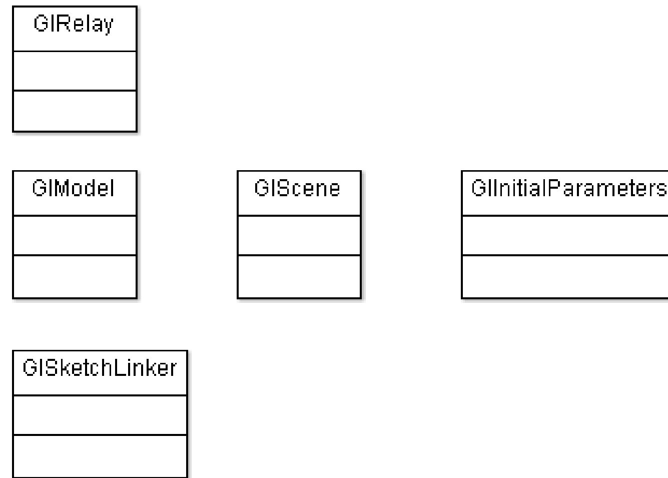


Figure 8.4: Class diagram of GroIMP module in Xplo.

ArchiTree in Xplo. Conversion is done in several steps:

- For each GroIMP graph node a global transformation matrix is computed.
- Check if the GroIMP graph is multi-scale.
- Traverse the GroIMP graph on the finest scale. For each step in this traversal, we get a pair of Nodes (linked by succession or branching edges), then a recursive function is called to reach the complex scales of the current nodes. ArchiNodes are created each time we reach a new GroIMP Node and a Map<Node, ArchiNode> is maintained to relate GroIMP nodes to ArchiNodes.
- Succession and Branching links between ArchiNodes are established when a common complex is encountered. Refinement (or Composition) links in ArchiTree must only be created for the first component Node(that is when a complex ArchiNode has no yet refinements (or components)).

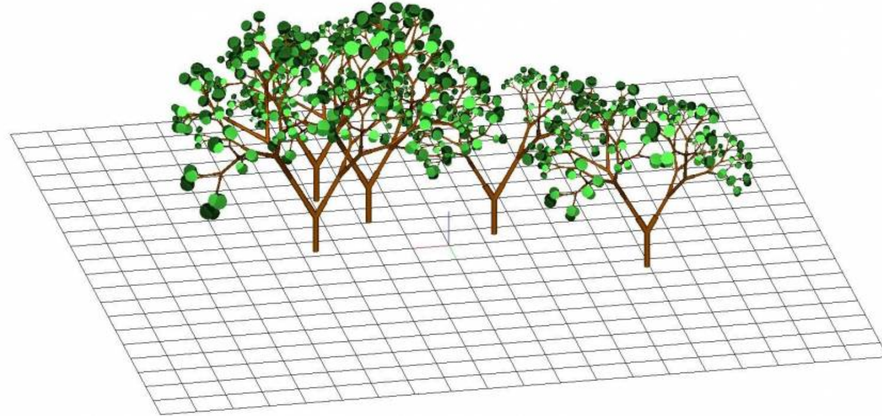


Figure 8.5: Screenshot of Xplo displaying GroIMP generated trees.

The conversion of some GroIMP node classes has been implemented, e.g. rotation nodes (RU,RL,RH), F (cylinder), Sphere. Type conversion hierarchy must be completed in jeeb.xplo.module.groimp.model.types. The function copyValues() must be overridden and must call super.copyValues() to transfer all member variable values in GroIMP nodes and their superclasses to ArchiNodes.

The usage of the interface is as follows:

- Start Xplo and run module GroIMP (Menu Project → New).
- A panel is displayed where user can select the folder with the XL script files, then set the number of simulation step (Age) and the output step frequency (Output Step = 1 means each simulation steps are saved), then press OK.

A screenshot of GroIMP generated plant architectures is shown in Figure 8.5.

Additional notes on the interface and GroIMP module in Xplo are found at http://amapstudio.cirad.fr/soft/xplo/private/groimp_module.

Bibliography

[1] OpenAlea MTG documentation. http://openalea.gforge.inria.fr/doc/vplants/newmtg/doc/_build/html/contents.html. Accessed: 16-02-2015.

[2] Xplo wiki. http://amap-dev.cirad.fr/projects/xplo/wiki. Accessed: 16-02-2015.

[3] *Proceedings of the 7th International Conference on Functional-Structural Plant Models*, Saariselkä, Finland, 2013. Finnish Society of Forest Science, Vantaa, Finland.

[4] H. Abelson and A. diSessa. *Turtle Geometry*. MIT Press, Cambridge, Massachusetts, 1982.

[5] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):990–1034, 2002.

[6] J. Adámek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories: The Joy of Cats*. Dover Publications, 2009.

[7] M. Aono and T. L. Kunii. Botanical tree image generation. *IEEE Computer Graphics and Applications*, 4:10–34, 1984.

[8] M. Aronoff and K. Fudeman. *What is Morphology?* Wiley-Blackwell, 2010.

[9] P. Balandier, A. Lacointe, X. LeRoux, H. Sinoquet, P. Cruiziat, and S. Le Dizès. SIMWAL: a structural-functional model simulating single walnut tree growth in response to climate and pruning. *Annals of Forest Science*, 57:571–585, 2000.

[10] G. A. Betz, C. Knappe, C. Lapierre, M. Olbrich, G. Welzl, C. Langebartels, W. Heller, H. Sandermann, and D. Ernst. Ozone affects Shikimate pathway transcripts and monomeric lignin composition in European beech (*Fagus sylvatica L.*). *European Journal of Forest Research*, 128:109–116, 2009.

[11] R. B. Bird, Armstrong R. C., and Hassager O. *Dynamics of Polymeric Liquids*. John Wiley, 1987.

[12] G. Boole. *The Mathematical Analysis of Logic*. Macmillan, Barclay, & Macmillan, London, 1847.

[13] G. Boole. *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities.* Dover Publications, New York, 1958.

[14] F. Boudon, C. Pradal, T. Cokelaer, P. Prusinkiewicz, and C. Godin. L-Py: an L-system simulation framework for modeling plant architecture development based on a dynamic language. *Frontiers in Plant Science*, 3(00076), 2012.

[15] Bart Braden. The surveyor's area formula. *The College Mathematics Journal*, 17(4):326–337, 1986.

[16] A. Brandt. Multiscale scientific computation: Review 2001. In T. J. Barth, T. Chan, and R. Haimes, editors, *Multiscale and Multiresolution Methods: Theory and Applications*, pages 3–96. Springer, Berlin, 2002.

[17] B. Breckling. An individual based model for the study of pattern and process in plant ecology: An application of object oriented programming. *Ecosys*, 4:241–254, 1996.

[18] A. G. Bromley. Charles Babbage's Analytical Engine, 1838. *IEEE Annals of the History of Computing*, 20:29–45, 1998.

[19] A. G. Bromley. Babbage's analytical engine plans 28 and 28a - the programmer's interface. *IEEE Annals of the History of Computing*, 22:5–19, 2000.

[20] G Buck-Sorlin, O. Kniemeyer, and W. Kurth. A model of poplar (Populus sp.) physiology and morphology based on relational growth grammars. In *Mathematical Modeling of Biological Systems, Volume II*, Modeling and Simulation in Science, Engineering and Technology, pages 313–322. Birkhäuser, Boston, 2008.

[21] H. E. Burkhart, K. D. Farrar, R. L. Amateis, and R.F. Daniels. Simulation of individual tree growth and stand development in loblolly pine plantations on cutover, site-prepared areas. *Virginia Polytechnic Institute and State University, Blacksburg, Pub.*, FWS-1-87, 1987.

[22] R. Car and M. Parrinello. Unified approach for molecular dynamics and density-functional theory. *Physical Review Letters*, 55:2471–2474, 1985.

[23] Magnus Carlsson. Monads for incremental computing. In *ACM SIGPLAN International Conference on Functional Programming*, pages 26–35. ACM, New York, USA, 2002.

[24] S. Chapman. Tercentenary of the calculating machine. *Nature*, 150:427–427, 1942.

[25] S. Chapman. *Fortran 95/2003 for Scientists & Engineers.* McGraw-Hill, 2007.

[26] N. Chiba, S. Ohkawa, K. Muraoka, and M. Miura. Visual simulation of botanical trees based on virtual heliotropism and dormancy break. *The Journal of Visualization and Computer Animation*, 5:3–15, 1994.

[27] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[28] N. Chomsky. *Syntactic Structures*. Mouton, Gravenhage, 1957.

[29] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[30] J. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19:547–554, 1976.

[31] Malte Clasen and Steffen Prohaska. Image-error-based level of detail for landscape visualization. In Reinhard Koch, Andreas Kolb, and Christof Rezk-Salama, editors, *Vision, Modeling and Visualization Workshop*, pages 267–274. Eurographics Association, 2010.

[32] W. F. Clocksin and C. S. Mellish. *Programming in Prolog: Using The ISO Standard*. Springer, Berlin, 2003.

[33] H. Cochard, S. Coste, B. Chanson, J. Guehl, and E. Nicolini. Hydraulic architecture correlates with bud organogenesis and primary shoot growth in beech (*Fagus sylvatica*). *Tree Physiology*, 25:1545–1552, 2005.

[34] H. Cochard and M. T. Tyree. Xylem dysfunction in quercus: vessel sizes, tyloses, cavitation and seasonal changes in embolism. *Tree Physiology*, 6:393–407, 1990.

[35] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In Michael F. Cohen, editor, *SIGGRAPH '98*, pages 115–122. ACM, 1998.

[36] A. Corradini, U. Montanari, and F. Rossi. Algebraic approaches to graph transformation: Part I. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1, chapter 3, pages 163–245. World Scientific, 1997.

[37] R. Cozzi. *The Modern RPG IV Language*. MC Press, USA, 2006.

[38] K. Culik and A. Lindenmayer. Parallel graph generating and graph recurrence systems for multicellular development. *International Journal of General Systems*, 3(1):53–66, 1976.

[39] P. de Reffye, C. Edelin, J. Françon, M. Jaeger, and C. Puech. Plant models faithful to botanical structure and development. In *Proceedings of SIGGRAPH 88*, pages 151–158, New York, NY, USA, 1988. ACM.

[40] Philippe Decaudin and Fabrice Neyret. Rendering forest scenes in real-time. In Alexander Keller and Henrik Wann Jensen, editors, *Eurographics Symposium on Rendering*, pages 93–102. Eurographics Association, 2004.

[41] B. Denby, I. Sundvor, M. Cassiani, P. de Smet, F. de Leeuw, and J. Horaálek. Spatial mapping of ozone and SO2 Trends in Europe. *Science of the Total Environment*, 408:4795–4806, 2010.

[42] Qingqiong Deng, Xiaopeng Zhang, Gang Yang, and Marc Jaeger. Multiresolution foliage for forest rendering. *Computer Animation and Virtual Worlds*, 21:1–23, 2010.

[43] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. Interactive visualization of complex plant ecosystems. In *Conference on Visualization '02*, VIS '02, pages 219–226. IEEE Computer Society, 2002.

[44] D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10:112–122, 1973.

[45] M. E. Dyer and H. E. Burkhart. Compatible crown ratio and crown height models. *Canadian Journal of Forest Research*, 17:572–574, 1987.

[46] W. E. *Principles of Multiscale Modeling.* Cambridge University Press, United Kingdom, 2011.

[47] W. E and B. Engquist. The heterogeneous multiscale methods. *Communications in Mathematical Sciences*, 1:87–132, 2003.

[48] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation: Part II. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1, chapter 4, pages 247–312. World Scientific, 1997.

[49] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: an algebraic approach. In *Proceedings IEEE Conference on Automata and Switching Theory '73*, pages 167–180. IEEE, 1973.

[50] 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.

[51] P. Eichhorst and W. J. Savitch. Growth functions of stochastic Lindenmayer systems. *Information and Control*, 45:217–228, 1980.

[52] J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1, chapter 1, pages 1–97. World Scientific, 1997.

[53] Jing Fan, YunYi Fan, TianYang Dong, and Lei Ji. Real-time information recombination of complex 3d tree model based on visual perception. *Science China Information Sciences*, 56(9):1–14, 2013.

[54] R. Feys. Boole as a logician. *Proceedings of the Royal Irish Academy*, 57:97–106, 1955.

[55] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics. Principles and Practice.* Addison-Wesley, Massachusetts, 1997.

[56] E. D. Ford, A. Avery, and R. Ford. Simulation of branch growth in the Pinaceae: Interactions of morphology, phenology, foliage productivity, and the requirement for structural support, on the export of carbon. *Journal of Theoretical Biology*, 146:15–36, 1990.

[57] G. Frege. *Begriffsschrift: Eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle, 1879.

[58] G. Frege. *Die Grundlagen der Arithmetik: Eine logisch mathematische Untersuchung über den Begriff der Zahl*. W. Koebner, 1884.

[59] D. Frijters and A. Lindenmayer. A model for the growth and flowering of Aster novaeangliae on the basis of table (1,0) L-systems. In Grzegorz Rozenberg and Arto Salomaa, editors, *L systems*, volume 15 of *Lecture Notes in Computer Science*, pages 24–52. Springer-Verlag, Berlin, 1974.

[60] T. Früh. Simulation of water flow in the branched tree architecture. *Silva Fennica*, 31:275–284, 1997.

[61] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH '93*, pages 247–254. ACM, 1993.

[62] T. Gabriele and M. Beyer. Amalgamated graph transformations and their use for specifying AGG – an algebraic graph grammar system. In *LNCS 776*, pages 380–394. Springer, 1994.

[63] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[64] Geoffrey Y. Gardner. Simulation of natural scenes using textured quadric surfaces. *ACM SIGGRAPH Computer Graphics*, 18(3):11–20, 1984.

[65] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97*, pages 209–216. ACM, 1997.

[66] V. L. Gavrikov and O. P. Sekretenko. Shoot-based three-dimensional model of young scots pine growth. *Ecological Modelling*, 88:183–193, 1996.

[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 analyzing plants with the AMAPmod software. In M. Michalewicz, editor, *Advances in Computational Life Sciences, Vol I : Plants to Ecosystems*, pages 53–84. CSIRO, Australia, 1997.

[69] J. F. Gouyet. *Branching in Nature*, volume 14 of *Centre de Physique des Houches*. Springer, France, 2001 edition, 2001.

[70] S. Griffon and F. de Coligny. AMAPstudio: An editing and simulation software suite for plants architecture modelling. *Ecological Modelling*, 290:3–10, 2014.

[71] J. Hanan. *Parametric L-systems and Their Application To the Modelling and Visualization of Plants*. PhD thesis, University of Regina, 1992.

[72] J. Hanan. Functional-structural modelling with L-systems: Where from and where to. In *Proceedings of the 7th International Conference on Functional-Structural Plant Models*, pages 1–3, Saariselkä, Finland, 2013. Finnish Society of Forest Science, Vantaa, Finland.

[73] D. W. Hann. An adjustable predictor of crown profile for stand-grown Douglas-fir trees. *Forest Science*, 45:217–225, 1999.

[74] J. L. Harper, B. R. Rosen, and J. White. *The Growth and Form of Modular Organisms*. The Royal Society, London, 1986.

[75] R. Hemmerling. *Extending the Programming Language XL to Combine Graph Structures with Ordinary Differential Equations*. PhD thesis, University of Göttingen, 2012.

[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] G. T. Herman and G. Rozenberg. *Developmental Systems and Languages*. North Holland, Amsterdam, 1975.

[78] E. Hobsbawm. *The Age of Revolution: 1789-1848*. Vintage, 1996.

[79] P. Hogeweg and B. Hesper. A model study on biomorphological description. *Pattern Recognition*, 6:165–179, 1974.

[80] H. Hollerith. *In Connection with the Electric Tabulation System Which Has Been Adopted by U.S. Government for the Work of the Census Bureau*. Dissertation, Columbia University, 1890.

[81] H. Honda. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology*, 31:331–338, 1971.

[82] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96*, pages 99–108. ACM, 1996.

[83] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH '97*, pages 189–198. ACM, 1997.

[84] M. H. Kalos and P. A. Whitlock. *Monte Carlo Methods*. Wiley-VCH, Weinheim, 2008.

[85] M-Z. Kang and P. de Reffye. A mathematical approach estimating source and sink functioning of competing organs. In 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*, pages 65–74, Berlin, 2007. Springer.

[86] L. Kari, G. Rozenberg, and A. Salomaa. L systems. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 5, pages 253–328. Springer-Verlag, 1997.

[87] S. Kellomäki and H. Strandman. A model for the structural growth of young Scots pine crowns based on light interception by shoots. *Ecological Modelling*, 80:237–250, 1995.

[88] R. Kerner, J. B. Winkler, J. W. Dupuy, M. Jürgensen, C. Lindenmayr, D. Ernst, and G. Müller-Starck. Changes in the proteome of juvenile European beech following three years exposure to free-air elevated ozone. *iForest*, 4:69–76, 2011.

[89] I. G. Kevrekidis, C. W. Gear, J. M. Hyman, P. G. Kevrekidis, O. Runborg, and C. Theodoropoulos. Equation-free, coarse-grained multiscale computation: Enabling microscopic simulators to perform system-level analysis. *Communications in Mathematical Sciences*, 1:715–762, 2003.

[90] A.J. Klar. Developmental choices in mating-type interconversion in fission yeast. *Trends in Genetics*, 8(6):208–213, 1992.

[91] O. Kniemeyer. *Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling.* PhD thesis, Brandenburg University of Technology Cottbus, 2008.

[92] O. Kniemeyer, G. Barczik, R. Hemmerling, and W. Kurth. Relational growth grammars - a parallel graph transformation approach with applications in biology and architecture. In *AGTIVE 2007*, pages 152–167, 2007.

[93] O. Kniemeyer, G. Buck-Sorlin, and W. Kurth. A graph grammar approach to artificial life. *Artificial Life*, 10(4):413–431, 2004.

[94] 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. *Berichte des Forschungszentrums Waldökosysteme der Universität Göttingen, Ser. B*, 38, 1994.

[95] W. Kurth. Proposal for a Research Grant - A Generic Functional-Structural Plant Model with Applications to the Development and Interaction of Young Forest Trees, Unpublished, 2013.

[96] Winfried Kurth, Ole Kniemeyer, and Gerhard Buck-Sorlin. *Relational growth grammars - A graph rewriting approach to dynamical systems with a dynamical structure*, volume 3566 of *Lecture Notes in Computer Science*, book section 5, pages 56–72. Springer Berlin Heidelberg, 2005.

[97] S. M. Lane. *Categories for the Working Mathematician.* Springer, New York, 1998.

[98] T. Lang. Rules for robot draughtsmen. *Geographical Magazine*, 42:50–51, 1969.

[99] W. Lenzen. Leibniz und die Boolesche Algebra. *Studia Leibnitiana*, 16:187–203, 1984.

[100] U. Leupold. Studies on recombination in Schizosaccharomyces pombe. *Cold Spring Harbor Symposia on Quantitative Biology*, 23:161–170, 1958.

[101] A. Lindenmayer. Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300 – 315, 1968.

[102] Aristid Lindenmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, 18:280–315, 1968.

[103] Yotam Livny, Soeren Pirk, Zhanglin Cheng, Feilong Yan, Oliver Deussen, Daniel Cohen-Or, and Baoquan Chen. Texture-lobes for tree modelling. *ACM Transactions on Graphics*, 30(4):53:1–53:10, 2011.

[104] Javier Lluch, Emilio Camahort, Jose Luis Hidalgo, and Roberto Vivo. A hybrid multiresolution representation for fast tree modeling and rendering. *Procedia Computer Science*, 1(1):485–494, 2010.

[105] S. L. Lohr. *Sampling: Design and Analysis.* Cengage Learning, Boston, 2009.

[106] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.

[107] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics.* Morgan Kaufmann Publishers, San Francisco, USA, 2003.

[108] B. B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman, San Francisco, USA, 1982.

[109] Dana Marshall, Donald S. Fussell, and A.T. Campbell. Multiresolution rendering of complex botanical scenes. In *Graphics Interface '97*, pages 97–104, 1997.

[110] E. Martin. The calculating machines. In M. Campbell-Kelly and W. F. Aspray, editors, *Charles Babbage Institute Reprint Series for the History of Computing*, volume 16. The MIT Press, London, England, 1992.

[111] J. Martin. *Applications Development without Programmers.* Prentice Hall, 1982.

[112] H. Maruyama, K. Tamura, and N. Uramoto. *XML and Java: Developing Web Applications.* Addison-Wesley Longman, Amsterdam, 2002.

[113] A. Mateescu and A. Salomaa. Aspects of classical language theory. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 4, pages 175–251. Springer-Verlag, 1997.

[114] A. Mateescu and A. Salomaa. Formal languages: an introduction and a synopsis. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 1, pages 1–39. Springer-Verlag, 1997.

[115] C. Maus, S. Rybacki, and A. M. Uhrmacher. Rule-based multi-level modeling of cell biological systems. *BMC Systems Biology*, 5(1):166, 2011.

[116] Nelson Max. Hierarchical rendering of trees from precomputed multi-layer z-buffers. In *Eurographics Workshop on Rendering Techniques*, pages 165–174. Springer-Verlag, London, UK, 1996.

[117] Robert B. McMaster. The geometric properties of numerical generalization. *Geographical Analysis*, 19(4):330–346, 1987.

[118] F. L. Menabrea. Sketch of the analytical engine invented by Charles Babbage. *Scientific Memoirs*, 3:666–, 1842.

[119] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop*, pages 157–168. Springer, 1998.

[120] Alexandre Meyer, Fabrice Neyret, and Pierre Poulin. Interactive rendering of trees with shading and shadows. In Steven J. Gortler and Karol Myszkowski, editors, *Eurographics Workshop on Rendering*, pages 183–196. Springer-Verlag, 2001.

[121] M. Mitchell. *Complexity: A Guided Tour*. Oxford University Press, U.S.A., 2011.

[122] Y. N. Moschovakis. What is an algorithm. In B. Engquist and W. Schmid, editors, *Mathematics Unlimited — 2001 and beyond*, pages 919–936. Springer, 2001.

[123] R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH 96*, pages 397–410. ACM, 1996.

[124] M. Nagl. On a generalization of Lindenmayer-systems to labelled graphs. In A. Lindenmayer and G. Rozenberg, editors, *Automata, Languages, Development*, pages 487–508. North Holland, 1976.

[125] Y. Ong. Multi-scale rule-based graph transformation using the programming language XL. In H. Ehrig, G. Engels, H-J. Kreowski, and G. Rozenberg, editors, *Graph Transformations. 6th International Conference, ICGT 2012*, Lecture Notes in Computer Science 7562, pages 417–419, Berlin, 2012. Springer.

[126] Y. Ong and W. Kurth. A design pattern in XL for implementing multiscale models, demonstrated at a fruit tree simulator. Abstract submitted to X International Symposium on Modelling in Fruit Research and Orchard Management, 2015, Montpellier, France.

[127] Y. Ong and W. Kurth. Incremental LOD for multi-resolution branching structures. Submitted to Eurographics Symposium on Geometry Processing, 2014.

[128] Y. Ong and W. Kurth. A graph model and grammar for multi-scale modelling using XL. In J. Gao, W. Dubitzky, C. Wu, M. Liebman, R. Alhaij, L. Ungar, A. Christianson, and X. Hu, editors, *2012 IEEE International Conference on Bioinformatics and Biomedicine Workshops*, pages 1–8, Philadelphia, USA, 2012. IEEE Computer Society.

[129] Y. Ong and W. Kurth. Developing multiscale simulation models using the software GroIMP. In J. Wittmann and D. K. Maretis, editors, *Simulation in den Umwelt- und Geowissenschaften. Workshop Osnabrück 2014*, pages 51–64, Aachen, 2014. Shaker Verlag.

[130] Y. Ong, K. Streit, M. Henke, and W. Kurth. An approach to multiscale modelling with graph grammars. *Annals of Botany*, 114:813–827, 2014.

[131] H. Opheim. Fast data reduction of a digitized curve. *Geo-Processing*, 2:33–40, 1982.

[132] G. O'Regan. *A Brief History of Computing*. Springer-Verlag, London, 2008.

[133] W. Palubicki, K. Horel, S. Longay, A. Runions, B. Lane, R. Mech, and P. Prusinkiewicz. Self-organizing tree models for image synthesis. *ACM Transactions on Graphics*, 28:1–10, 2009.

[134] J. Perttunen, R. Sievänen, E. Nikinmaa, H. Salminen, H. Saarenmaa, and J. Väkevä. LIGNUM: a tree model based on simple structural units. *Annals of Botany*, 77:87–98, 1996.

[135] A. M. Petritan, B. von Lüpke, and I. C. Petritan. Effects of shade on growth and mortality of maple (Acer pseudoplatanus), ash (Fraxinus excelsior) and beech (Fagus sylvatica) saplings. *Forestry*, 80(4):397–412, 2007.

[136] C. Pradal, S. Dufour-Kowalski, F. Boudon, C. Fournier, and C. Godin. OpenAlea: a visual programming and component-based software platform for plant modelling. *Functional Plant Biology*, 35(10):751–760, 2008.

[137] P. Prusinkiewicz. Graphical applications of L-systems. In *Proceedings on Graphics Interface 86/Vision Interface 86*, pages 247–253, Toronto, Canada, 1986. Canadian Information Processing Society.

[138] P. Prusinkiewicz. Modeling and visualization of biological structures. In *Proceeding of Graphics Interface '93*, pages 128–137, 1993.

[139] P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Měch. Visual models of plant development. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 9, pages 535–597. Springer-Verlag, 1997.

[140] P. Prusinkiewicz, M. Hammel, and E. Mjolsness. Animation of plant development. In *SIGGRAPH 93*, pages 351–360. ACM, 1982.

[141] P. Prusinkiewicz and J. Hanan. *Lindemayer Systems, Fractals, and Plants*. Springer Verlag, Berlin, 1989.

[142] P. Prusinkiewicz and L. Kari. Subapical bracketed L-systems. In J. Cuny, H. Ehrig, G. Engles, and G. Rozenberg, editors, *Grammars and their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 550–564. Springer-Verlag, 1996.

[143] P. Prusinkiewicz, A. Lindenmayer, and J. Hanan. Developmental models of herbaceous plants for computer imagery purposes. In *Proceedings of SIGGRAPH 88*, pages 141–150, New York, NY, USA, 1988. ACM.

[144] P. Prusinkiewicz and R. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, New York, 1990.

[145] Przemyslaw Prusinkiewicz. Graphical applications of L-systems. In *Graphics Interface '86*, pages 247–253. Canadian Information Processing Society, 1986.

[146] K. Reumann and A. P. M. Witkam. Optimizing curve segmentation in computer graphics. In *International Computing Symposium*, pages 467–472. North-Holland Publishing Company, 1974.

[147] J. Rossignac and P. Borrel. Multi-resolution 3d approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics*, IFIP Series on Computer Graphics, pages 455–465. Springer, 1993.

[148] A. Rostand-Mathieu, P-H. Cournède, and P. de Reffye. A dynamical model of plant growth with full retroaction between organogenesis and photosynthesis. *ARIMA*, 4:101–107, 2006.

[149] M. K. Roy and D. G. Dastidar. *COBOL Programming.* Tata McGraw-Hill Education, 1989.

[150] G. Rozenberg and A. Salomaa. *The Mathematical Theory of L-systems.* Academic Press, New York, 1980.

[151] G. Rozenberg and A. Salomaa, editors. *Beyond Words*, volume 3 of *Handbook of Formal Languages.* Springer-Verlag, 1997.

[152] B. Russell. *The Principles of Mathematics.* Cambridge University Press, 1903.

[153] A. Salomaa. *Formal Languages.* Academic Press, New York, 1973.

[154] V. Sarlikioti, P. H. B. de Visser, and L. F. M. Marcelis. Exploring the spatial distribution of light interception and photosynthesis of canopies by means of a functional-structural plant model. *Annals of Botany*, 107(5):875–883, 2010.

[155] M. Scalfi, M. Troggio, P. Piovani, S. Leonardi, G. Magnaschi, G. G. Vendramin, and P. Menozzi. A RAPD, AFLP and SSR linkage map, and QTL analysis in European beech (*Fagus sylvatica L.*). *Theoretical and Applied Genetics*, 108:433–441, 2004.

[156] H. Schildt. *Java: The Complete Reference.* Mcgraw-Hill Osborne Media, 2014.

[157] R. Schober. *Ertragstafeln wichtiger Baumarten.* J. D. Sauerländer's Verlag, Frankfurt am Main, 1995.

[158] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *Proceedings of SIGGRAPH '92*, pages 65–70. ACM, 1992.

[159] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57:713–723, 1938.

[160] W. Shi and C. Cheung. Performance evaluation of line simplification algorithms for vector generalization. *The Cartographic Journal*, 43(1):27–44, 2006.

[161] H. Simmons. *An Introduction to Category Theory.* Cambridge University Press, 2011.

[162] A. R. Smith. Plant, fractals, and formal languages. In *Proceedings of SIGGRAPH 84*, pages 1–10. ACM, 1984.

[163] B. Stern and P. Nurse. Fission yeast pheromone blocks S-phase by inhibiting the G1 cyclin Bp34cdc2 kinase. *EMBO Journal*, 16(3):534–544, 1997.

[164] P. S. Stevens. *Patterns in Nature*. Little, Brown and Company, 1974.

[165] J. Strobel. *Die Atmung der verholzten Organe von Altbuchen (Fagus sylvatica L.) in einem Kalk- und einem Sauerhumusbuchenwald*. Dissertation, University of Göttingen, Germany, 2004.

[166] B. Stroustrup. *The C++ Programming Language: Special Edition*. Addison Wesley, 2000.

[167] A. L. Szilard and R. E. Quinton. An interpretation for D0L systems by computer graphics. *The Science Terrapin*, 4:8–13, 1974.

[168] A. Takenaka. A simulation model of tree architecture development based on growth response to local light environment. *Journal of Plant Research*, 107:321–330, 1994.

[169] D. W. Thompson. *On Growth and Form*. Cambridge University Press, 1952.

[170] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[171] A. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London B*, 237:37–72, 1952.

[172] J. J. Tyson. Modeling the cell division cycle: cdc2 and cyclin interactions. *Proceedings of the National Academy of Sciences USA*, 88(16):7328–7332, August 1991.

[173] S. Ulam. On some mathematical properties connected with patterns of growth of figures. In *Proceedings of Symposia on Applied Mathematics*, volume 14, pages 215–224. American Mathematical Society, 1962.

[174] M. Visvalingam and J. D. Whyatt. Line generalization by repeated elimination of points. *The Cartographic Journal*, 30:46–51, 1993.

[175] A. Warshel and M. Levitt. Theoretical studies of enzymic reactions: Dielectric, electrostatic and steric stabilization of the carbonium ion in the reaction of lysozyme. *Journal of Molecular Biology*, 103:227–249, 1976.

[176] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *SIGGRAPH '95*, pages 119–128. ACM, 1995.

[177] T. Yamada-Inagawa, A.J.S. Klar, and J.Z. Dalgaard. Schizosaccharomyces pombe switches mating type by the synthesis-dependent strand-annealing mechanism. *Genetics*, 177:255–265, 2007.

[178] H. Yan, M. Z. Kang, P. de Reffye, and M. Dingkuhn. A dynamic, architectural plant model simulating resource-dependent growth. *Annals of Botany*, 93:591–602, 2004.

[179] Z. Zhao and A. Saalfeld. Linear-time sleeve-fitting polyline simplification algorithms. In *Auto-Carto 13, ACSM ASPRS Annual Convention and Exposition*, volume 5, pages 214–223, 1997.